Université de Montréal

Approche probabiliste pour l'analyse de l'impact des changements dans les programmes orientés objet

par
Aymen Zoghlami

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des arts et des sciences
en vue de l'obtention du grade de Maître ès (M.Sc.)
en informatique

Juin, 2011

Université de Montréal
Faculté des arts et des sciences

Ce mémoire intitulé :

Approche probabiliste pour l'analyse de l'impact des changements dans les programmes orientés objet

Présenté par :
Aymen Zoghlami

a été évalué par un jury composé des personnes suivantes :

Pierre L'Ecuyer, président-rapporteur
Houari Sahraoui, directeur de recherche
Bruno Dufour, co-directeur
Julie Vachon, membre du jury

Résumé

Nous proposons une approche probabiliste afin de déterminer l'impact des changements dans les programmes à objets. Cette approche sert à prédire, pour un changement donné dans une classe du système, l'ensemble des autres classes potentiellement affectées par ce changement. Cette prédiction est donnée sous la forme d'une probabilité qui dépend d'une part, des interactions entre les classes exprimées en termes de nombre d'invocations et d'autre part, des relations extraites à partir du code source. Ces relations sont extraites automatiquement par rétro-ingénierie. Pour la mise en œuvre de notre approche, nous proposons une approche basée sur les réseaux bayésiens. Après une phase d'apprentissage, ces réseaux prédisent l'ensemble des classes affectées par un changement. L'approche probabiliste proposée est évaluée avec deux scénarios distincts mettant en œuvre plusieurs types de changements effectués sur différents systèmes. Pour les systèmes qui possèdent des données historiques, l'apprentissage a été réalisé à partir des anciennes versions. Pour les systèmes dont on ne possède pas assez de données relatives aux changements de ses versions antécédentes, l'apprentissage a été réalisé à l'aide des données extraites d'autres systèmes.

Mots-clés : Analyse d'impact de changement, maintenance, modèle probabiliste, réseaux bayésiens

Abstract

We study the possibility of predicting the impact of changes in object-oriented code using bayesian networks. For each change type, we produce a bayesian network that determines the probability that a class is impacted given that another class is changed. Each network takes as input a set of possible relationships between classes. We train our networks using historical data. The proposed impact-prediction approach is evaluated with two different scenarios, various types of changes, and five systems. In the first scenario, we use as training data, the changes performed in the previous versions of the same system. In the second scenario training data is borrowed from systems that are different from the changed one. Our evaluation showed that, in both cases, we obtain very good predictions, even though they are better in the first scenario.

Keywords: Change Impact Analysis, maintenance, probabilistic models, bayesian networks

Table des matières

Chapitre 1 1
Introduction générale 1
1.1 Contexte 1
1.2 Problématique 2
1.3 Contribution 3
1.4 Structure du mémoire 5
Chapitre 2. 6
État de l'art 6
2.1 Introduction 6
2.2 Analyse de l'impact basée sur la traçabilité 6
2.3 Analyse d'impact basée sur les dépendances 10
2.4 Analyse d'impact des changements et réseaux bayésiens 19
2.5 Synthèse et conclusion 23
Chapitre 3 27
Construction des modèles d'analyse de l'impact des changements 27
3.1 Introduction 27
3.2 Rappel sur les réseaux bayésiens 28
3.2.1 Définition 28
3.2.2 Fonctionnement 30
3.2.3 Mise en œuvre des réseaux bayésiens : l'inférence 33
3.3 Construction de la structure des réseaux 33
3.4 Paramétrage des réseaux bayésiens 42
3.5 Conclusion 47
Chapitre 4. 48
Mise en œuvre et évaluation 48
4.1 Introduction 48
4.2 Mise en œuvre 48
4.3 Le protocole de validation 53
4.3.1 Collecte des données 53

4.3.2 Mesure de la performance dans un contexte probabiliste 55
4.3.3 Évaluation de l'approche proposée 56
4.4 Prédictions intra-système 57
4.4.1 Modification d'une classe (CC) 61
4.4.2 Suppression d'une classe (DC) 62
4.4.3 Ajout d'une méthode (AM) 63
4.4.4 Modification d'une méthode (CM) 66
4.4.5 Suppression d'une méthode (DM) 68
4.4.6 Ajout d'un attribut (AF) 70
4.4.7 Suppression d'un attribut (DF) 71
4.5 Prédictions inter-systèmes 73
4.5.1 Modification d'une classe (CC) 75
4.5.2 Suppression d'une classe (DC) 77
4.5.3 Ajout d'une méthode (AM) 78
4.5.4 Modification d'une méthode (CM) 81
4.5.5 Suppression d'une méthode (DM) 84
4.5.6 Surpression d'un attribut (DF) 85
4.6 Conclusion 87
Chapitre 5. 89
Conclusion 89
5.1 Rétrospectives et contributions 89
5.2 Perspectives futures 90

Liste des tableaux

Tableau 1: synthèse des travaux sur la prédiction de l'impact 24
Tableau 2: synthèse des travaux sur l'analyse de l'impact 25
Tableau 3: les systèmes utilisés pour l'évaluation..... 54
Tableau 4: précisons et rappels intra-système..... 58
Tableau 5: précisions et rappel inter-systèmes..... 74
Tableau 6: synthèse des résultats des prédictions..... 88

Liste des figures

Figure 1 : l'analyse de l'impact basée sur la traçabilité entre les artefacts [42] 7
Figure 2: exemple de graphe d'appel de contrôle 16
Figure 3: réseau d'impact de changement. 21
Figure 4: exemple de réseau bayésien [27] 29
Figure 5: exemple de matrice des liens (Xerces 1.3.1) 36
Figure 6: réseau bayésien pour le changement des classes 38
Figure 7: réseau bayésiens pour la suppression des classes 38
Figure 8: réseau bayésien pour l'ajout de méthodes 39
Figure 9: réseau bayésien pour le changement de méthodes 39
Figure 10: réseau bayésien pour la suppression de méthodes 40
Figure 11: réseau bayésien pour l'ajout d'attribut 41
Figure 12: réseau bayésien pour la suppression d'attribut 41
Figure 13: processus d'apprentissage des réseaux 42
Figure 14: graphes des changements et fermeture transitive 43
Figure 15: exemple de fichier d'apprentissage 46
Figure 16: mise en œuvre des réseaux bayésiens 49
Figure 17: partitionnement flou pour le nombre d'invocations 52
Figure 18: ensembles flous du nombre d'invocations 53
Figure 19: allure idéale pour la précision et le rappel 60
Figure 20: JFreeChart: précision et rappel pour le changement de classe (CC) 61
Figure 21: XERCES: précision et rappel pour le changement de classe (CC) 62
Figure 22: JFreeChart: précision et rappel pour le changement de classe (DC) 63
Figure 23: JFlex: précision et rappel pour l'ajout de méthode (AM) 64
Figure 24: XERCES: précision et rappel pour l'ajout de méthode (AM) 65
Figure 25: OpenJmail: précision et rappel pour l'ajout de méthode (AM) 66
Figure 26: EIRC: précision et rappel pour le changement de méthode (CM) 67
Figure 27: JFlex: précision et rappel pour le changement de méthode (CM) 68
Figure 28: EIRC: précision et rappel pour la suppression de méthode (DM) 69
Figure 29: JFreeChart: précision et rappel pour la suppression de méthode (DM) 70
Figure 30: JFlex: précision et rappel pour l'ajout d'un attribut (AF) 71

Figure 31: EIRC: précision et rappel pour la suppression d'un attribut (DF) 71
Figure 32: XERCES: précision et rappel pour la suppression d'un attribut (DF) 72
Figure 33: JFreeChart: précision et rappel pour la suppression d'un attribut (DF) 73
Figure 34: XERCES: précision et rappel pour le changement d'une classe (CC) 76
Figure 35:JFreeChart: précision et rappel pour le changement d'une classe (CC) 76
Figure 36: JFreeChart: précision et rappel pour la suppression d'une classe (DC) 77
Figure 37: EIRC: précision et rappel pour la suppression d'une classe (DC) 78
Figure 38: EIRC: précision et rappel pour l'ajout d'une méthode (AM)..... 79
Figure 39: OpenJmail: précision et rappel pour l'ajout d'une méthode (AM)..... 80
Figure 40: JFreechart: précision et rappel pour l'ajout d'une méthode (AM) 80
Figure 41: XERCES: précision et rappel pour l'ajout d'une méthode (AM)..... 81
Figure 42:EIRC: précision et rappel pour la modification d'une méthode (CM) 82
Figure 43: OpenJMail: précision et rappel pour la modification d'une méthode (CM)..... 83
Figure 44: JFlex: précision et rappel pour la modification d'une méthode (CM) 84
Figure 45: XERCES: précision et rappel pour la suppression d'une méthode (DM) 85
Figure 46: JFreeChart: précision et rappel pour la suppression d'une méthode (DM) 85
Figure 47: EIRC: précision et rappel pour la suppression d'un attribut (DF) 86
Figure 48: XERCES: précision et rappel pour la suppression d'un attribut (DF) 87
Figure 49: représentation du système ArgoUML avec la métaphore de la ville 92

Je tiens à dédier ce travail à :

Mes parents qui ont fait des sacrifices énormes pour que je devienne l'Homme que je suis. De plus ils n'ont pas cessé de me soutenir durant toute ma scolarité.

Mon frère Sahbi et mes sœurs Hana et Hayet pour leurs conseils et amours inconditionnels.

Les nouveaux venus dans notre petite famille : Ma belle sœur Sabeh, mon beau frère Sofien et les petites Brahim, Imene et Khalil. Vous apportez que du bonheur et de la joie.

Mes amis qui m'encouragent et qui rendent mes journées pleines de joie et d'enthousiasme.

Je vous dédie tous ce travail et je vous promets que je ne vais jamais épargner d'efforts afin d'être toujours à la hauteur de vos attentes. Je vous aime.

Aymen

Remerciements

Je remercie tout d'abord mes directeurs Pr Houari Sahraoui et Pr Bruno Dufour pour avoir accepté de diriger ce travail. Je les remercie pour leur patience et leur disponibilité dont ils ont fait preuve. Je suis reconnaissant au Professeur Houari pour m'avoir initié à la recherche et a fait de sorte que j'aime la quête permanente de nouvelles informations. Quant au Professeur Bruno, son apport dans ce mémoire a été considérable et j'ai admiré collaborer avec lui vu sa disponibilité et son sens de perfection dans tous les détails. Je les remercie également pour la confiance qu'ils m'ont témoignée tout au long de ce mémoire.

J'adresse pareillement mes sincères remerciements aux membres de jury pour l'honneur qu'ils ont fait en participant à ce jury et d'avoir bien voulu apporter leurs conseils et leurs jugements sur ce travail.

Je remercie les membres de laboratoire de génie logiciel GEODES. En effet, c'est pour moi un plaisir et un honneur d'en faire partie. À chaque fois que j'avais besoin d'aide, tous les membres étaient prêts à partager leurs savoir-faire chacun dans son domaine. Ainsi, Simon, Marouane, Stéphane, Jamel, Arbi, Dorsaf et tous les membres m'ont aidé chacun d'une façon. Par ailleurs, l'aspect humain a été fondamental, car nos relations dépassent la camaraderie et les barbecues et les sorties entre « Géodésiens » peuvent en témoigner.

Je remercie aussi tous les membres du Département d'informatique et de recherche opérationnelle et notamment les professeurs avec lesquels j'ai fait des cours durant ma maîtrise.

Finalement, je remercie tous les professeurs qui ont contribué à ma formation et le gouvernement tunisien qui a participé au financement de ce mémoire de maîtrise et à tous ceux qui ont contribué de près ou de loin à sa réalisation.

Chapitre 1
Introduction générale

Dans ce chapitre introductif, nous présentons, dans le contexte général de la maintenance, l'analyse de l'impact des changements dans les programmes à objets. Par la suite, nous présentons la problématique, les limites des approches existantes et notre contribution pour pallier les problèmes identifiés. Enfin, nous présentons la structure de ce mémoire de maîtrise.

1.1 Contexte

La maintenance est la phase la plus coûteuse du cycle de vie du logiciel. Ceci peut, entre autres, s'expliquer par le fait qu'elle est la phase la plus longue qui ne s'achève qu'avec le retrait du logiciel. Par ailleurs, le ratio du coût de la maintenance par rapport au coût total du logiciel est passé de 40 à 60 % dans les années 80 à plus de 80% au début de l'an 2000 [52]. Le coût élevé de la maintenance pousse les chercheurs et les industriels à se focaliser sur cette phase du cycle de vie d'un système afin d'essayer de la comprendre pour mieux la contrôler. Pour y arriver, il existe des techniques telles que la visualisation qui sert à montrer les tendances générales du système d'une façon intuitive [41]. Par ailleurs selon les lois d'évolution des logiciels, comme indiqué dans [44], un système doit être continuellement changé. Autrement, il apportera moins de satisfaction dans son environnement. En effet, durant la phase de maintenance, les programmeurs sont amenés à faire des changements aux systèmes pour de multiples raisons. Par exemple, les changements réalisés peuvent permettre de répondre à de nouveaux besoins des utilisateurs. On parle dans ce cas, de maintenance perfective. La maintenance peut aussi consister à apporter des changements afin de prévenir les bogues (préventive), de s'adapter à un nouvel environnement (adaptative), ou pour corriger des erreurs (corrective). Ainsi, les logiciels doivent subir plusieurs changements pour prolonger la durée de leur vie.

1.2 Problématique

Apporter des changements aux systèmes existants s'avère indispensable. Cependant, un changement, même élémentaire, peut avoir des conséquences inattendues sur le fonctionnement global des systèmes. Par exemple, modifier la visibilité d'un attribut peut avoir des résultats non attendus sur le fonctionnement du système. Ces conséquences sont plus néfastes si l'on n'arrive pas à déterminer les éléments qui sont affectés par le changement réalisé. Une fois ces éléments détectés, ils doivent être modifiés par les mainteneurs afin de garantir le bon fonctionnement du système. Ce besoin d'identifier les éléments affectés par un changement constitue l'objectif de la tâche d'analyse de l'impact de changements.

L'analyse de l'impact de changements tente de déterminer les parties à modifier pour accommoder un changement. Les coûts énormes engendrés par les changements non contrôlés ont amené les chercheurs à s'intéresser aux techniques d'analyse d'impact des changements depuis longtemps. Par exemple, Weinberg [69] liste les erreurs de type logiciel qui ont coûté le plus. Ces erreurs sont souvent dues uniquement à un changement dans un chiffre. Un peu moins de vingt ans plus tard, il était toujours difficile de prédire l'impact du passage à l'an 2000 sur les systèmes informatiques. Ce problème, connu sous le nom d'Y2K, a incité plusieurs organisations à inclure l'analyse de l'impact de changement explicitement dans le processus de développement et de maintenance des logiciels [7]. Afin de réaliser cette tâche, plusieurs approches furent proposées. On peut les classer principalement en deux catégories : la première catégorie consiste à appliquer des techniques d'analyse d'impact sur des modèles et des documents tels que ceux décrivant l'architecture [64] ou à partir des besoins des utilisateurs [20]. La deuxième catégorie s'intéresse plutôt aux dépendances dans le code source afin d'identifier les parties du système qui doivent possiblement être modifiées.

Les approches existantes offrent aux programmeurs des outils pour effectuer l'analyse de l'impact de changement. Cependant, elles présentent des inconvénients. En effet, les modèles utilisés dans la première catégorie ne correspondent pas souvent aux systèmes existants étant donné que la mise à jour des documents de conception est rarement faite dans l'industrie. En plus, la manière d'implémenter les fonctionnalités exprimées dans les documents de conceptions est différente d'un programmeur à un autre.

Ceci rend ardue la tâche de détection de l'impact des changements. Par ailleurs, les approches qui se limitent à utiliser les dépendances du code source peuvent être plus ou moins efficaces à cause des problèmes liés à l'identification de ces dépendances d'une manière efficace [74]. Les techniques de « slicing », qui sont généralement utilisées à cette fin, sont coûteuses et peuvent avoir une précision assez bonne. Ces dernières consistent principalement à découper le programme en blocs d'instructions et d'identifier celui qui est mis en cause par le changement. Le découpage est généralement basé sur un graphe de dépendances entre éléments du logiciel. Les arcs entre les nœuds du graphe correspondent soit à des dépendances d'utilisation (lecture ou écriture de données), soit à des dépendances de contrôle si le résultat de l'exécution d'une instruction d'un élément modifie l'exécution d'un autre élément. La précision dans l'identification en utilisant les techniques de « slicing » est généralement bonne. L'ensemble des éléments affectés est déduit par la fermeture transitive du graphe de dépendance. L'ensemble, ainsi trouvé, peut inclure certain faux-positif, c.-à-d., des éléments qui ne sont pas réellement affectés.

L'objectif de ce mémoire est de pallier les problèmes cités ci-dessus en proposant une approche probabiliste qui est fondée sur la rétro-ingénierie des dépendances entre les classes du système. Par ailleurs, l'aspect non déterministe de la propagation des changements nous a encouragés à utiliser des réseaux bayésiens afin de prédire l'ensemble des classes affectées d'un système suite à un changement.

1.3 Contribution

Le but de ce travail d'offrir aux concepteurs et aux mainteneurs un moyen de prédire les conséquences des changements qu'ils comptent effectuer sur des programmes à objets. En effet, comme l'affirment Pfleeger et Bohner [53], « plus on comprend un changement, mieux on peut le contrôler et donc minimiser son risque ». Pour y parvenir, nous proposons une approche probabiliste.

Le choix d'une approche probabiliste est justifié par deux arguments. D'une part, l'analyse de l'impact se fait avant que le système ne soit réellement modifié et donc affecté. Ceci permet de réduire le coût et offre aux mainteneurs le choix entre plusieurs scénarios de maintenance selon l'ensemble des classes prédites comme affectées. D'autre part, avec cette approche probabiliste, nous essayons d'offrir un compromis entre deux

types d'approches. La première catégorie utilise des techniques d'analyse comme le « slicing » qui offre une bonne précision, mais dont le coût d'analyse est élevé [70]. Par contre, la deuxième catégorie, qui utilise des règles génériques et donc ne procède pas à une analyse élaborée, donne des résultats qui sont moins précis [9].

Pour atteindre notre objectif, nous optons pour une solution qui combine les avantages des approches basées sur les règles génériques et celles qui se focalisent sur l'analyse du code source après modification. En effet, notre approche consiste à analyser le code source pour extraire uniquement les dépendances entre classes sans effectuer des analyses d'impact couteuses. Ces relations peuvent être les liens définis dans les diagrammes de classes d'UML (Unified Modeling Language) [59] tels que l'héritage, les associations ou les agrégations. De plus, nous nous intéressons aux détails d'invocations entre les classes. Avec ces liens, notre approche couvre aussi bien les dépendances statiques que les dépendances dynamiques exprimées sous la forme d'appels entre les classes. Ces données collectées sont utilisées comme entrée à un système de prédiction. Le système de prédiction est lui basé sur des règles. Contrairement aux travaux où ces règles sont génériques, les nôtres sont obtenus à partir de données historiques.

Pour la mise en œuvre de notre approche, nous avons choisi d'utiliser les réseaux bayésiens pour déterminer la probabilité qu'une classe soit affectée suite à un changement effectué sur une autre classe du système. Ces réseaux sont entraînés à partir des données décrivant des situations réelles d'impact. Dans notre approche le terme «classe affectée » signifie que la classe en question va déclencher une erreur au moment de la compilation. L'intérêt et l'explication des choix des réseaux bayésiens vont être plus détaillés dans le troisième chapitre.

La validation de notre approche se fait sur plusieurs systèmes réels en deux volets. Premièrement, nous essayons de prédire l'impact d'un changement effectué sur un système à partir des données historiques des changements provenant de versions antérieures du même système. Ce cas de figure est utile dans le contexte où les programmeurs disposent d'informations relatives à l'évolution de ce logiciel (changements et leurs impacts). Le deuxième volet de la validation considère le scénario dans lequel la prédiction de l'impact de changement d'un système est basée sur des données provenant d'autres systèmes. Ce

cas de figure est intéressant quand l'historique des changements d'un système n'est pas disponible ou si le système vient d'être créé.

L'approche proposée, en plus de nous permettre de prédire les changements, peut aussi s'avérer très utile pour déterminer le scénario de changements le moins coûteux parmi un ensemble de choix possible. En effet, le classement des classes affectées selon leurs probabilités de changements aide les programmeurs à estimer l'effort de maintenance requis pour gérer les changements qu'ils comptent effectuer.

1.4 Structure du mémoire

Le reste du mémoire est organisé comme suit. Le deuxième chapitre présente un survol des principaux travaux en relation avec l'analyse de l'impact des changements et l'application des réseaux bayésiens dans ce domaine. La description détaillée de notre approche est présentée dans le troisième chapitre. Les résultats de l'évaluation de notre approche sont discutés dans le quatrième chapitre. Enfin, le dernier chapitre est consacré à la récapitulation des idées principales introduites dans ce mémoire, ainsi qu'à la description de quelques perspectives d'amélioration de l'approche proposée.

Chapitre 2.
État de l'art

2.1 Introduction

Nous présentons dans ce chapitre un survol des travaux existants sur l'analyse de l'impact des changements. Nous présentons en première partie les études qui se sont intéressées à l'analyse de l'impact des changements basée sur la traçabilité. Ensuite, et dans le même contexte que notre approche, nous allons présenter des travaux relatifs à l'analyse de la propagation des changements suivant les dépendances dans le code source. Enfin, nous clôturons ce chapitre par une présentation des travaux qui utilisent, comme notre approche, les réseaux bayésiens pour l'analyse de l'impact des changements selon différentes perspectives.

2.2 Analyse de l'impact basée sur la traçabilité

Diverses définitions sont utilisées afin de présenter la traçabilité. Celle qui est la plus couramment utilisée dans l'analyse de l'impact de changement est celle proposée par Arnold [4]. Ce dernier définit la traçabilité comme « la capacité à tracer entre des artefacts logiciels produits et modifiés pendant le cycle de vie de produit logiciel ». L'analyse de traçabilité, comme illustré dans la Figure 1, se concentre donc sur les dépendances entre les phases de cycle de vie d'un logiciel à partir de l'identification des besoins, de l'analyse des exigences et de la conception du code jusqu'à la génération du code et de la maintenance. L'analyse d'impact qui utilise la traçabilité se focalise sur les dépendances entre des éléments appartenant à des phases différentes pour identifier les artefacts logiciels qui doivent être modifiés suite à un changement. Par ailleurs, les documents qu'on utilise dans ce genre d'approche sont généralement en langue naturelle (exemple : les besoins des utilisateurs) et/ou semi-formels tels que les diagrammes. Par conséquent, les modèles

d'analyse d'impact de changement qui se basent sur la traçabilité sont le plus souvent manuels ou semi-automatiques.

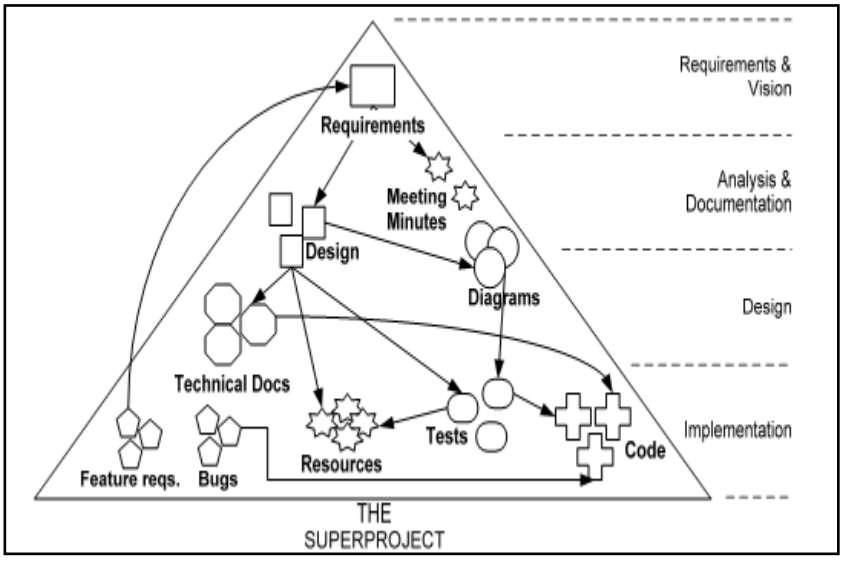

Figure 1 : l'analyse de l'impact basée sur la traçabilité entre les artefacts [42]

Un des premiers travaux qui se sont intéressés à l'analyse de l'impact des changements en se basant sur la traçabilité entre les différents artefacts du logiciel est celui élaboré au sein des laboratoires du Rome Air Developpement Center [58]. Le résultat de ce travail est l'élaboration du prototype ALICIA (Automated Life Cycle Impact Analysis). Ce prototype a été introduit afin de gérer l'analyse de l'impact de changement pour une base de données de modèles au sein du standard MIL-STD-2167 destiné aux applications militaires. Des règles de traçabilité décrivent le passage d'une phase à une autre du cycle de vie d'un document. L'analyse de l'impact des changements se réalise par la suite en se basant sur les informations enregistrées durant le développement. La tâche d'ALICIA est de prendre en considération le point de départ de l'analyse de l'impact et d'identifier automatiquement à travers les règles de traçabilités déjà définies les artefacts qui vont être affectés. À la fin de cette analyse, le prototype met à jour la base de données des modèles afin d'assurer la cohérence de cette base.

Hassan et al. [23] présentent une approche en quatre étapes afin de déterminer l'impact des changements effectués au niveau de l'architecture sur le code source. La première étape consiste à élaborer le modèle formel ASCM (Architectural Software

Components Model) pour décrire l'architecture du système. Ce modèle sert à présenter les composants qui constituent l'architecture du système ainsi que les relations entre eux. Les auteurs proposent ensuite une topologie des changements qui peuvent être appliqués à ces composants. Chaque changement est défini par son nom, son opération, ses pré-conditions, ses post-conditions et ses invariants. Après avoir présenté un modèle pour décrire l'architecture du système et l'ensemble des changements qui peuvent y être appliqués, les auteurs s'intéressent à identifier la propagation du changement voulu au niveau architectural. Pour y parvenir, ils optent pour un système de règles définies par des experts. Enfin et afin de détecter l'impact sur le code source, les auteurs couplent le modèle ASCM avec le modèle SCSM (Software Component Sructural Model). Ce dernier permet de présenter du code source écrit en différents langages. ASCM et SCSM sont définis comme des graphes et l'analyse de l'impact de changement se fait via une fonction qui fait la projection de chaque élément du modèle architectural sur un ou plusieurs éléments du modèle structurel. Ainsi, le calcul de l'impact de changement se fait sur le modèle présentant l'architecture et son effet sur les parties du code se détecte en utilisant la traçabilité entre les nœuds du ASCM et ceux du SCSM.

Göknil [20] présente un travail qui s'intéresse à la traçabilité entre les différentes exigences présentées par les utilisateurs. En effet, l'approche proposée se base sur un méta-modèle où les besoins sont reliés par des relations tels que « Raffine », « Exige », « Est en conflit » et « Contient ». Ces relations servent par la suite à l'élaboration de différentes règles de propagation de changements. L'analyse de l'impact des changements fait la différence entre les modifications effectuées au niveau des exigences, et celles au niveau des relations entre ces exigences. Par exemple, si les programmeurs comptent supprimer un besoin R1 alors R2 est affecté dans le cas où la relation reliant R1 et R2 est de type « contient ». Par contre R2 n'est pas affecté si R1 « raffine », « exige » ou « est en conflit avec » R2. L'objectif de cette technique est de permettre de détecter l'impact des changements réalisés dans les besoins des utilisateurs sur l'architecture du système en traçant les liens entre un besoin d'utilisateur et les composants qui vont le satisfaire.

Wong et Cai [72] présentent une autre technique qui se base sur un réseau de contraintes appelé ACN (Augmented Constraint Network). Les auteurs avancent l'idée que

le diagramme de classes d'UML ne prend pas en considération des données relatives à l'environnement et qui peuvent influencer les choix des programmeurs quand ils décident d'apporter des changements au système existant. Ainsi, ils définissent un ACN qui représente un réseau de contraintes extrait à partir de diagramme de classes UML. En effet, un ACN est composé de variables qui représentent les dimensions intervenantes dans la conception. Ces variables peuvent prendre des valeurs selon l'environnement et sont régies par des contraintes. Les contraintes servent à exprimer des relations entre des classes, des modules, des aspects et des facteurs relatifs à l'environnement comme la taille de la mémoire. L'approche de l'analyse de l'impact peut être résumée comme suit. La première étape consiste à calculer à partir du graphe de l'ACN, les composantes fortement connexes. Ces composantes représentent les sous-réseaux d'ACN qui définissent un ensemble d'actions à accomplir pour réaliser une tâche particulière. Ensuite, le programmeur indique la classe à changer. L'approche indique les classes affectées c.-à-d. celles qui sont reliées à cette classe dans l'ACN. Cependant, le poids d'impact n'est pas identique entre ces classes, mais il est proportionnel au nombre d'apparitions de chaque classe dans un sous-réseau contenant la source du changement. En plus, ce poids dépend aussi de la distance entre le nœud en question et le nœud auquel les programmeurs veulent apporter des changements. Les dix éléments dont le poids est le plus élevé sont proposés comme l'ensemble des parties affectées du système. Par ailleurs, les auteurs utilisent les données historiques, si elles sont disponibles, pour appliquer une approche hybride. Ainsi, ils définissent le degré de confiance comme le nombre de fois où chaque élément est changé simultanément dans le passé avec la source de changement. Ce degré multiplie les poids des changements calculés dans la première phase afin de les mettre à jour. Les résultats de cette approche ont été comparés aux résultats obtenus avec les techniques basées sur l'exploration des données historiques. Dans les premières versions, l'approche proposée donne de meilleurs résultats. Cependant, avec l'augmentation du nombre des versions disponibles, les techniques de la fouille de données deviennent meilleures, que l'approche qui utilise l'ACN. Par contre, les résultats obtenus avec l'approche hybride donnent de meilleurs résultats en termes de rappel, de précision et de F-mesure. Les définitions de ces trois mesures sont données dans la section 4.3.2.

Lock et Kotonya [46] introduisent l'aspect probabiliste dans l'analyse de l'impact de changement relatif aux besoins d'utilisateurs. Les auteurs identifient un ensemble d'inconvénients qui réduisent l'efficacité des approches existantes comme l'incomplétude des données historiques ou le fait qu'on ne prenne pas en considération l'incertitude qui existe lors de l'analyse de l'impact de changements. Pour pallier ces problèmes, les auteurs proposent une approche en deux étapes pour intégrer les probabilités dans l'analyse de l'impact de changement. La première étape consiste à utiliser une des approches se basant sur la traçabilité déjà définies dans la littérature. L'objectif de cette approche est de générer un graphe où les nœuds représentent les différents besoins des utilisateurs. Les arcs définissent les relations d'impacts entre les différents nœuds. La deuxième étape consiste à attribuer un poids à chaque arc indiquant la probabilité de propagation. Cette probabilité est attribuée selon les résultats de l'impact des changements semblables dans les données historiques. Par la suite, les auteurs modifient cette probabilité en y intégrant deux facteurs : le degré avec lequel les développeurs pensent que les informations qu'ils ont fournies sont complètes et le degré de certitude des développeurs en leurs choix concernant la propagation. La dernière partie de ce travail est consacrée aux techniques qui peuvent permettre de rendre cette approche applicable à grande échelle. En effet, les auteurs recommandent des solutions pour réduire la taille du graphe résultant de la première étape: la détection des chemins cycliques dans le graphe, la suppression des parties du graphe dont la probabilité est inférieure à un seuil et le filtrage des données selon une fonctionnalité précise pour avoir les données qui sont en relation avec ce besoin d'utilisateur.

2.3 Analyse d'impact basée sur les dépendances

Plusieurs approches utilisent des dépendances extraites du code source afin d'analyser l'impact de changement. Lee et Offut [45] automatisent ce processus en présentant un ensemble d'algorithmes qui s'intéressent aux changements apportés au code source. Les changements expriment des modifications faites au niveau des classes ou des méthodes. L'approche se base sur trois types de graphes.

- **Le graphe de dépendance des données au sein d'une méthode** (Intra-Method Data Dependency Graph) : c'est un graphe où les nœuds représentent des éléments d'une méthode et les arcs indiquent les dépendances entre ces éléments. Les pondérations des arcs indiquent le sens de propagation des changements.
- **Le graphe de dépendance des données entre des méthodes**: c'est un graphe qui regroupe les graphes qui représentent les dépendances au sein de chaque méthode. Les nœuds sont reliés par des arcs dont le poids indique le degré d'impact. Le degré d'impact varie de 0 à 3. Ce degré est égal à 0 si le nœud de départ n'a aucun effet sur le nœud d'arrivée, à 1 si le nœud d'arrivé est affecté mais ne propage pas l'impact à d'autres nœuds, à 2 si le nœud n'est pas affecté mais va propager le changement à d'autres nœuds et à 3 si le nœud d'arrivée est affecté par un changement dans le nœud de départ et va propager l'effet vers d'autres nœuds.
- **Le graphe de dépendance du système orienté objet**: c'est un graphe S= (N, E, R, C, W) où N est l'ensemble des nœuds qui représentent des classes, E est l'ensemble des arcs qui représentent les dépendances entre les classes, R est une fonction qui associe un type de relations entre les classes (héritage, agrégation, appel) à chaque arc, C est la fonction qui attribue le type d'impact aux arcs et W est la fonction qui attribue des poids de l'impact entre les classes aux arcs.

Les auteurs supposent que chaque changement exprimé par les utilisateurs peut être transformés en un triplet (*c, m, t*) avec *c* indiquant la classe à changer, *m* la méthode à modifier et *t* le changement à effectuer. Une fois le changement identifié, l'outil proposé construit les trois types de graphes précités puis calcule l'impact de changement au niveau de la classe à changer et sur tout le système. Ce calcul est réalisé par l'application d'une fermeture transitive sur les graphes. Cette fermeture transitive va donner comme résultat les nœuds qu'on peut visiter à partir du nœud de départ où le changement s'effectue.

Une autre façon d'analyser l'impact des changements est d'essayer de comprendre la relation entre la structure d'un système et son comportement face à des changements. Chaumun et al. [9] s'intéressent à cet aspect. Ils commencent par définir un modèle d'analyse d'impact de changement. Ce modèle prédit l'ensemble des classes affectées par un changement d'une façon booléenne. La détermination des classes affectées dépend du

changement à effectuer et des liens qui relient les classes comme l'association, l'héritage, l'agrégation et l'invocation. Comme les premières expérimentations de ce modèle ont été réalisées sur des systèmes en C++, les auteurs ont aussi ajouté un autre lien « Friendship » pour prendre en considération les spécificités du langage. Les résultats de ces expérimentations sur un système industriel ont permis de déduire que le système en question ne nécessite pas de la maintenance suite à des changements relatifs aux signatures des méthodes. Un autre résultat plus important est la détection de relation de corrélation entre la métrique WMC (Weighted Methods per Class) qui exprime la somme de complexité des méthodes d'une classe et la fréquence de ses changements.

Les résultats de ce travail ont incité les mêmes auteurs à s'intéresser à la relation entre les métriques de conception et la propagation des changements [11]. L'objectif de ce travail est d'étudier la corrélation entre des métriques de couplage, définies par Chidamber and Kemerer dans [13], connues sous le nom de (C&K), et l'impact de changement. Les auteurs ont ajouté les métriques suivantes dérivées des métriques C&K afin de mieux représenter les facteurs qui influencent la sensibilité des systèmes par rapport aux changements :

- NOC* (Number Of Children in sub-tree): cette métrique est utile dans le cas où le changement ne se limite pas aux enfants directs d'un nœud, mais se propage dans tout le sous-arbre.
- CBO_NA (Coupling Between Objects No Ancestors): les auteurs utilisent cette métrique, car ils pensent qu'il serait judicieux d'exclure les ancêtres des classes lors du calcul du couplage vu que le changement ne se propage pas des enfants vers les parents.
- CBO_IUB (Coupling Between Objects Is Used By): le couplage classique ne prend pas en compte le sens de l'utilisation des classes. Cette métrique se limite aux classes qui utilisent la classe à modifier.
- CBO_U (Coupling Between Objects Using): cette métrique représente les classes qui sont utilisées par a classe à changer.

Les résultats ont permis d'affirmer la forte relation entre le couplage et la propagation des changements. Par conséquent, les auteurs confirment l'hypothèse que les propriétés de

conception ont une influence sur les effets que les changements peuvent avoir sur les systèmes.

Kabaili, Keller et Lustman [39] ont utilisé le modèle d'analyse d'impact de changement, défini par Chaumun et al. [9] , pour tester si les métriques de cohésion peuvent être considérées comme indicateurs de la capacité du système à tolérer les changements qu'on lui apporte. Un ensemble de six types de changements sur les classes, les méthodes et les attributs ont été considérés. De plus, les auteurs ont choisi deux métriques de cohésion (LCC : Loose Class Cohesion et LCOM : Lack of COhesion in Methods). Les premiers résultats obtenus ont démontré qu'il n'y avait pas de relation entre la cohésion et la propagation des changements. Ce résultat surprenant a poussé les auteurs à inspecter le code manuellement et ils ont fini par conclure que les métriques choisies ne reflètent pas souvent la cohésion qui existe réellement dans le code source.

Haney [22] présente une méthode pour identifier les effets qu'un système peut subir suite à un changement. La méthode présentée est basée sur une matrice où les lignes et les colonnes correspondent aux différents modules du système. Les cases de la matrice représentent un nombre entre 0 et 1 indiquant une probabilité d'impact. Ainsi la case de matrice $M_{(i,j)} = 0,7$ indique qu'un changement dans le module i va avoir un impact sur le module j avec une probabilité égale à 0,7. Cette approche est simpliste, car les probabilités sont déterminées d'une façon subjective. Ces probabilités sont par la suite utilisées pour déterminer le coût de maintenance. En effet, à chaque module correspond un ensemble de changements à réaliser pour maintenir le bon fonctionnement du système. Les probabilités sont modifiées par la suite selon l'expérience déjà acquise suite à l'historique des changements. Une des contributions principales de ce travail est l'introduction du terme « ripple effect ». En effet, ce dernier signifie que l'analyse de l'impact de changement ne se limite pas aux classes directement affectées, mais considère aussi l'effet que cet ensemble affecté puisse avoir sur le reste du système. Ainsi, le « ripple effect » étend la portée de l'analyse de l'impact des changements des classes affectées directement aux classes indirectement affectées par le changement.

Briand, Jurgen et Lounis ont réalisé une étude empirique [8] afin de vérifier la relation entre les mesures de couplage et le « ripple effect ». L'étude a été élaborée sur un

système dont ils possèdent un ensemble de versions et une liste des changements qui ont été appliqués. La démarche consiste à calculer une moyenne de couplage entre chaque couple de classes. Cette moyenne est le résultat de la combinaison de trois métriques de cohésion : PIM (Polymorphism Method Invocations), CBO' (CBO sans inclure les classes reliées par l'héritage) et INAG (INdirect AGgregation). Une fois que les classes sont triées selon leurs degrés de cohésion, les auteurs calculent la corrélation entre ce classement et les classes affectées en ayant recours à l'historique des changements. Les résultats de cette étude empirique ont montré que malgré le fait que les métriques définies dans la littérature ne permettaient pas de détecter tous les impacts de changement, elles peuvent être un bon indicateur pour déterminer les classes qui sont susceptibles de changer dans les versions futures.

L'analyse de l'impact de changement s'avère aussi une approche intéressante pour la gestion des tests de maintenance. Ryder et Tip [60] proposent une technique qui considère un ensemble de tests et un ensemble de changements atomiques. Le but de cette approche est de déterminer l'ensemble des tests possiblement affectés par des changements atomiques et l'ensemble des changements qui ont un effet sur un test en particulier. Ainsi, on peut savoir si le changement effectué compromet le fonctionnement du système et connaître les tests qui sont nécessaires pour détecter un impact de changement. Ces changements peuvent être décomposés en des changements atomiques, qui s'appliquent aux classes, aux méthodes et aux attributs. Pour réaliser l'analyse de l'impact des changements, l'approche proposée par Ryder suppose l'existence du système initial P et la version modifiée P' de P qui est syntaxiquement correcte et qu'on peut compiler. L'approche a été implémentée dans l'outil Chianti [57]. Ren, Chesley et Ryder ont utilisé Chianti pour la détection des changements qui causent des problèmes dans les programmes Java [56]. En effet, lors du passage d'une version à une autre, les programmeurs effectuent plusieurs changements. Cette multitude de changements rend ardue la tâche de détection de la cause d'un bogue dans le système après qu'il soit changé. Les auteurs ont aussi implémenté l'outil Crisp qui crée des versions intermédiaires du système que les mainteneurs ont modifié. Crisp se base sur Chianti afin de décomposer le changement entre les versions en un ensemble de changements atomiques. En introduisant à chaque fois

un sous-ensemble de ces changements sur le système, Crisp génère une version intermédiaire que le programmeur peut tester. Si les résultats montrent qu'il y a un problème alors les programmeurs peuvent déduire que c'est dû au dernier sous ensemble de changements intégré au système.

Hattori et al. [24] implémentent l'outil « Impala ». Cet outil utilise des techniques statiques basées sur le graphe d'appel du système à modifier. À partir d'un changement identifié par son type et par l'entité concernée, l'outil crée une représentation du programme. Cette dernière est composée d'une part d'entités qui peuvent être des classes, des méthodes ou des attributs et d'autre part par des relations qui peuvent avoir des significations telles que : « instanceOf », «isSuperclass », « invokeInterface ». Une fois cette présentation du système établie, l'outil Impala fait appel à un algorithme de calcul d'impact selon le changement à effectuer. L'outil offre la possibilité d'indiquer le degré de calcul d'analyse qui peut se limiter aux entités qui ont reliées directement ou bien élargir cette recherche à 2, 3 ou plusieurs niveaux. Cette option fut le sujet d'une étude empirique pour déterminer l'effet de ce degré dans les résultats en terme de précision et de rappel et pour évaluer l'outil présenté.

Linda Badri et Mourad Badri proposent une approche [5] pour prédire l'impact de changement qui est basé sur le CCG (Control Call Graph) illustré dans la Figure 2. En effet, les graphes d'appel utilisés dans des travaux antérieurs présentent des inconvénients dans la mesure où ils ne prennent pas en considération certains aspects tels que l'impossibilité de détecter l'ordre d'appel des méthodes ou les conditions sous lesquelles l'appel peut s'effectuer. Ainsi, les auteurs suggèrent que les CCG puissent offrir de meilleurs résultats en termes d'analyse d'impact de changement. Le CCG est un graphe orienté où les nœuds représentent des instructions, des points de décisions ou des blocs d'instructions. Les arcs représentent un transfert de contrôle. La démarche pour la prédiction de l'impact peut se résumer comme suit : une première étape où on analyse le code source et l'on crée des CCG de méthodes à travers une analyse statique et une seconde étape de prédiction. Soit M la méthode à modifier, l'ensemble IM (Impacted Methods) est l'union de l'ensemble des méthodes qui appellent M, l'ensemble des méthodes appelées par M après le changement et les méthodes appelées après l'exécution de M. L'évaluation

de cette approche montre que les résultats obtenus sont des ensembles de méthodes de cardinalité inférieure aux méthodes utilisant les graphes d'appel.

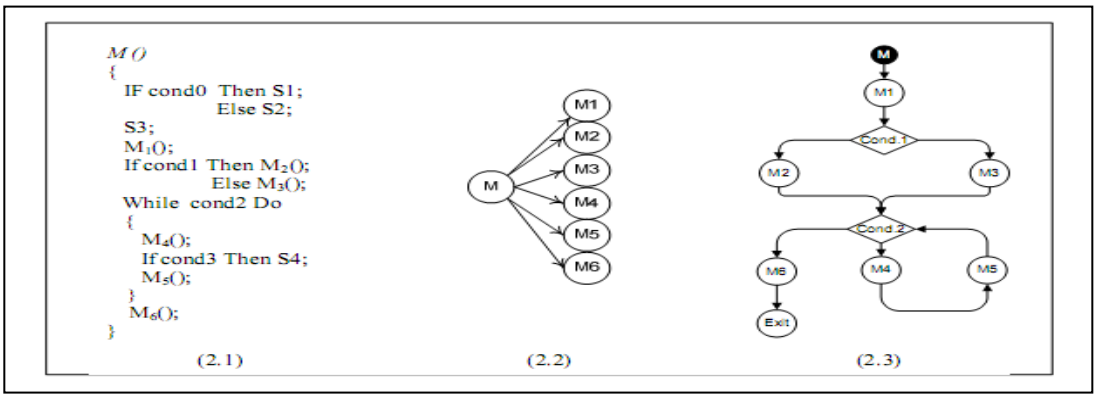

Figure 2: exemple de graphe d'appel de contrôle

Un autre aspect pour l'analyse de l'impact des changements consiste à s'intéresser aux co-changements (les entités qui changent simultanément.) Plusieurs travaux se sont penchés sur cette idée. Par exemple, Jashki, Zafarani et Bagheri [38] proposent une méthode qui peut être présentée comme suit. Premièrement, les répertoires des versions sont étudiés afin de déterminer les fichiers qui sont proches en termes de modifications. Le résultat de cette étape est une matrice qui indique le degré de dépendance entre ces fichiers. La dernière étape de l'approche consiste à trouver des groupes (cluster) des fichiers. La tâche de groupement a été réalisée selon cinq algorithmes différents pour en choisir le meilleur. Les résultats ont montré que l'algorithme DBSCAN (Density-Based Spatial Clustering of Applications with Noise) semble le mieux adapté pour la détection des parties des programmes qui sont affectées par les changements.

Les approches présentées précédemment sont dites statiques, car elles sont basées sur l'analyse de la structure du système à partir du code source. On trouve dans la littérature une autre catégorie d'approche d'analyse d'impact de changements qui utilise des méthodes dynamiques. Parmi les contributions dans ce domaine on peut notamment mentionner le travail de Law et Rothermel [43] qui utilise les traces d'exécution. Ces traces sont obtenues après l'exécution du programme en ayant recours à une instrumentation qui enregistre l'entrée et la sortie de chacune des méthodes. Ensuite, ces traces sont compressées par l'algorithme SEQUITUR qui les transforme en des

descriptions conformes à une grammaire. Cette dernière peut être présentée sous forme d'un graphe orienté et acyclique. Enfin, l'algorithme PathImpact parcourt ce graphe pour déterminer les nœuds affectés suite à un changement. Ce parcours se fait à partir du nœud à modifier pour détecter les méthodes qui appellent la méthode modifiée et on fait des retours en arrière dans le graphe pour détecter les méthodes dont le comportement peut être affecté suite à un appel de la méthode à modifier.

Orso, Apiwattanapong et Harrold [50] présentent une approche qui utilise les exécutions collectées auprès des utilisateurs du système. La méthode fait la couverture des exécutions dans un vecteur où la case $X_{(i,j)} = 0$ si dans l'exécution i, la méthode j n'est pas appelée et 1 sinon. Ce vecteur sert de point de départ pour l'algorithme « CoverageImpact » afin de calculer l'impact de changement dans une méthode m. La démarche consiste en trois étapes. La première est la détection des exécutions qui passent par la méthode m en question (dont la valeur = 1 dans les vecteurs). La deuxième étape est le calcul des parties (slicing) du programme à partir de m et l'ajout des méthodes qui s'y trouvent. La dernière étape procède à l'intersection entre les méthodes couvertes (dont la valeur =1) et les méthodes obtenues dans la deuxième phase.

Ces deux techniques dynamiques ont fait l'objet d'une étude empirique réalisée par Orso, Apiwattanapong, Law, Rothermel, et Harrold [51]. Cette étude empirique a pour objectif la comparaison entre les deux techniques en termes de précision et du coût en termes d'espace et de temps. Les expérimentations étaient effectuées sur trois systèmes et les résultats peuvent être résumés comme suit : en terme de précision, la méthode de Law est plus précise, car elle utilise les traces et non les couvertures. Par contre elle est plus coûteuse en termes d'espace mémoire et de temps d'exécution. En effet, le temps dans l'approche d'Orso est constant pour la mise à jour du vecteur à chaque entrée de méthode. Par contre, ce temps dépend de la taille de la trace dans l'approche de Law. Concernant la complexité spatiale, la méthode d'Orso est linéaire par rapport au nombre de méthodes (un bit pour chaque méthode) alors que ça dépend de la taille des traces chez Law.

Apiwattanapong, Orso et Harrold [2] proposent une méthode qui essaie de réduire les inconvénients des deux méthodes dynamiques précédentes. Pour cela, les auteurs cherchent à identifier les relations du type « Execute-After ». La nouveauté dans ce travail

est que cette relation n'est pas faite au niveau des exécutions des méthodes, mais plutôt entre les instructions des méthodes. Ainsi, l'information utilisée consiste en la première et dernière exécution de la méthode. Cette simplification offre une réduction de complexité sans pour autant réduire l'information sur l'impact. Par exemple, une méthode x est considérée comme affectée par un changement dans une méthode y si l'ordre de l'apparition de la première instruction de x précède la dernière apparition de y dans la trace d'exécution. La mise en œuvre de cette méthode a été faite par l'outil EAT (Execute-After Tool) qui est composé de trois parties : un module qui se charge d'instrumenter le code afin de détecter les points d'entrée et de sortie des méthodes, des moniteurs dont le rôle est de mettre à jour les indices des éventements des méthodes tout au long de l'exécution et un module qui calcule l'ensemble des méthodes affectées. Les résultats empiriques de cette approche montrent que cette approche offre un meilleur compromis entre la précision et le temps d'exécution que les deux techniques présentées par Orso et al. [50] [51].

Huang et Song [25] ont proposé une technique qui tente d'améliorer les techniques dynamiques existantes en termes de précision et de coût. En termes de précision, elle utilise les dépendances de données pour identifier les méthodes réellement affectées. En effet, l'algorithme présenté sélectionne les traces d'exécution. Celles qui ne contiennent pas la méthode à changer sont éliminées dans une phase préalable au calcul de l'impact de changement. De plus, l'algorithme utilise les flots de données afin d'avoir une meilleure analyse. Ainsi, les méthodes appelées après la méthode changée sont exclues de l'ensemble des méthodes affectées s'il n'y a pas de dépendances de données entre elles et la méthode changée. En termes de coût, l'algorithme présenté évite les méthodes redondantes. Cependant, les résultats n'ont pas été validés par un outil qui implémente cette approche.

Les approches précitées ne tiennent pas compte de la nature du système et des changements à effectuer. De plus elles sont généralement dédiées aux changements des méthodes. A partir de ces constatations, Huang et Song [26] nous présentent une approche réservée aux systèmes orientés objet et qui tient compte à la fois des dépendances du système et de la nature du changement. Ainsi ils ont donné une liste de changements atomiques relatifs aux méthodes et aux attributs. Le calcul de l'impact se réalise avec la technique « Execute After » présentée par les mêmes auteurs [25] . Cette analyse utilise

des graphes de dépendances et des traces pour identifier l'ordre de l'exécution des méthodes. Les graphes de dépendances sont générés afin de détecter les différentes variantes des interactions entre les méthodes (passage de paramètre, accès à un attribut…). L'étude comparative avec d'autres techniques d'analyse d'impact a montré que la prise en compte des dépendances entre les entités du système et de la nature du changement à appliquer peut améliorer la précision des résultats obtenus.

Par ailleurs, on trouve des méthodes qui cherchent à analyser l'impact de changement avec une approche basée sur la recherche documentaire comme celle définie par Poshyvanyk, Marcus, Ferenc et Gyimóthy [54]. La méthode présentée utilise la mesure de couplage pour déterminer l'impact de changement. Contrairement aux autres techniques, le couplage n'est pas déterminé en termes de nombre d'appels entre les classes, mais plutôt identifié en appliquant des techniques de recherche documentaire et plus précisément l'indexation sémantique latente. En effet, le programmeur identifie le niveau de granularité qu'il voudrait analyser (classe ou méthode). Le code source est considéré comme un document texte, et la technique d'indexation permet de calculer un degré de distribution des mots par document et générer une matrice qui correspond à différentes valeurs de couplage. Les auteurs utilisent la même démarche que celle de Briand [8] pour déterminer la relation entre les valeurs de couplage calculées et l'impact de changement. Les valeurs de couplages sont corrélées avec les classes affectées pour le système Mozilla. Les résultats montrent que cette approche qui sert à calculer le couplage peut renseigner sur l'impact de changement et donne de meilleurs résultats que les approches existantes.

2.4 Analyse d'impact des changements et réseaux bayésiens

Un réseau bayésien est défini par un graphe acyclique. Des variables aléatoires sont attribuées à ces nœuds et les arcs définissent une relation de causalité entre les nœuds parents et les nœuds fils (voir section 3.2 pour une définition détaillée). Conscients des bénéfices que peuvent apporter les réseaux bayésiens dans l'analyse de l'impact des changements, plusieurs chercheurs les ont utilisés afin de gérer l'incertitude relative à la propagation des changements. Par exemple, Tang et al. [64] ont utilisé des réseaux

bayésiens pour l'analyse de l'impact des changements au niveau architectural. En effet, dans cet article, les auteurs présentent le modèle AREL (Architecture Rationale and Element Linkage) qui est utilisé pour présenter la relation entre les décisions à apporter et les éléments de conception qui dépendent de ces choix de modification d'architecture. Par la suite ce modèle fut transposé en des réseaux bayésiens où les nœuds sont les éléments du AREL et les arcs sont les relations de dépendances entre ces nœuds. Les réseaux bayésiens ont fait l'objet d'une étude de cas qui consiste en une application du domaine bancaire. Les besoins fonctionnels ont été définis et traduits en AREL et ensuite en réseaux bayésiens dont les tables de probabilités sont définies par des spécialistes. Ces réseaux ont été utilisés sous trois volets. Le premier volet consiste à diagnostiquer les raisons d'un phénomène observé : en partant du nœud fils qui exprime le résultat observé, les programmeurs identifient les causes probables qui sont les noeuds pères du nœud en questions. L'aspect prédiction consiste à estimer les effets qu'un changement de besoins ou dans l'environnement pourrait avoir sur les éléments de l'architecture en identifiant les relations de causalité dans le réseau bayésien. Enfin, les réseaux bayésiens sont utilisés pour résoudre une combinaison des deux premiers volets.

Abdi, Lounis et Sahraoui [1] se sont intéressés à la compréhension des facteurs réels qui sont responsables de l'impact de changement et de son évolution. Des métriques de conception et d'implémentation sont étudiées afin de comprendre leurs effets sur les systèmes. Les auteurs ont proposé une approche probabiliste qui utilise les réseaux bayésiens pour déterminer l'impact des changements comme illustré dans la Figure 3. Les nœuds des réseaux bayésiens sont donc décomposés en deux catégories :

- Les nœuds d'entrée : représentant des métriques que les auteurs ont classifiées en des métriques de conception et d'implémentation.
- Les nœuds intermédiaires regroupent les métriques selon les deux types cités ci-dessus.

Une fois le réseau bayésien mis en place, les auteurs ont pu procéder à des scénarios de variation des valeurs des métriques pour identifier leurs effets sur l'impact du changement. En appliquant trois scénarios différents, les auteurs remarquent que la probabilité de l'impact augmente avec l'élévation des valeurs des métriques de conception comme

CBONA (Coupling Between Object No Ancestors) et CBOU (Coupling Between Object Using) et elle diminue en diminuant ces valeurs de métriques. Le dernier résultat montre que la réduction de la valeur de la métrique de conception AMMIC (Ancestors Method–Method Import Coupling) réduit légèrement l'impact.

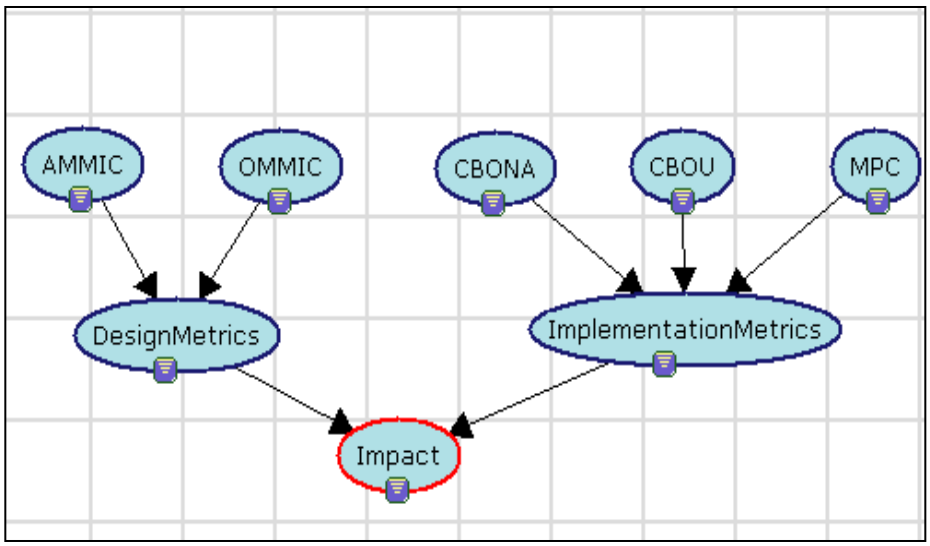

Figure 3: réseau d'impact de changement.

Mirarab [47] a proposé une approche qui utilise deux sources d'information: les métriques de dépendances et les données historiques. Les métriques de dépendances reflètent le degré de liaison entre les paquetages du système exprimé en nombre d'appels entre les classes de chaque paquetage. Ces dépendances peuvent correspondre à des appels méthodes ou à des utilisations de variables. Pour la détection de ces dépendances, les auteurs utilisent des techniques statiques. Le deuxième type d'information qui est les données historiques représente des données relatives aux co-changements dans le passé. Bien que cette information ne présente pas la propagation des changements, les auteurs font l'hypothèse que les éléments changés en même temps dans le passé ont de fortes chances de changer de la même façon dans les versions futures. Une fois ces données collectées, la deuxième étape consiste à construire des réseaux bayésiens pour prédire l'impact des changements. Les réseaux bayésiens utilisés dans cette approche sont de trois types :

- BDM (Bayesien Dependency Model)
- BHM (Bayesian History Model)
- BDHM (Bayesian Dependency and History Model).

Le processus débute par la création du « common model ». C'est un graphe orienté de tous les éléments avec des arcs pondérés qui indiquent le poids de dépendances entre les éléments. L'étape suivante consiste à éliminer les cycles de ce graphe en essayant de garder les arcs qui ont le poids le plus fort. Ainsi on obtient le réseau bayésien, qu'on nomme BDM (puisqu'il est basé sur les dépendances). Celui-ci peut être utilisé directement pour l'inférence ou enrichi par des données relatives à l'historique des changements dans le cadre des BDHM. Par ailleurs, on peut utiliser seulement les données historiques et appliquer l'inférence sur les données du BHM. Bien que la structure des graphes ne soit pas déclarée explicitement, les auteurs ont procédé à l'évaluation de leurs approches en utilisant le système Azureus. La validation du modèle proposé a été réalisée en comparant les prédictions des trois types de réseaux et d'une méthode qui attribue des valeurs de prédiction au hasard. Le seuil qui permet de prédire qu'une classe est affectée par un changement a été fixé à 40%. Ainsi toutes les classes prédites avec une probabilité de changement supérieure à ce seuil sont ajoutées à l'ensemble des classes affectées. Par ailleurs, les auteurs ont réalisé des expérimentations et ils ont conclu qu'un seuil de 50% serait plus adéquat. Par ailleurs, les résultats en termes de précision et de rappel ont démontré que le BDHM peut le mieux prédire les changements.

Enfin, Zhou et al. [76] présentent un modèle de réseaux bayésiens qui prédit la probabilité des co-changements entre les entités du système. Les informations servant de point d'entrée aux réseaux sont extraites à partir des données historiques et des dépendances dans le code et sont essentiellement : l'âge du changement, son auteur, la fréquence des changements, l'objectif des changements et d'autres informations qui servent à prédire les classes potentiellement affectées par un changement. Le processus de cette approche est divisé en trois étapes. La première consiste à collecter les données à partir des dépôts de contrôle de versions. Ces données sont transformées en une base de données de changements. La deuxième étape consiste à extraire les facteurs qui peuvent influencer les changements détectés tels que l'auteur qui les a effectués, l'entité du système

qui a été modifiée ou la date du changement vu que les auteurs adhèrent à l'idée que les entités changées depuis longtemps deviennent plus stables. Ces facteurs forment ensuite les nœuds des réseaux bayésiens qui vont être, durant la dernière phase du processus, entraînés pour pouvoir prédire l'impact des changements. L'évaluation de cette approche réalisée sur ArgoUML et Azureus a montré que 75% des prédictions correspondent à la réalité pour le premier système et 86% pour le second. De plus, parmi les classes qui sont réellement affecté une proportion de 53% et 71%, respectivement pour ArgoUML et Azureus, a été prédites par cette approche. Parmi les limites identifiées par les auteurs, nous trouvons le fait que ces résultats soient affectés par des facteurs de subjectivité, car ils ont procédé au choix des parties responsables de l'apprentissage.

2.5 Synthèse et conclusion

L'étude de l'impact de changement est une activité fondamentale dans le génie logiciel car elle peut servir à planifier des changements, à les mettre en place et à prévoir ou détecter leurs effets sur le système et essayer de les réduire.

Diverses méthodes ont été présentées dans la littérature pour ce volet de la maintenance. Ce chapitre présente un aperçu des travaux relatifs à l'analyse de l'impact des changements. On a décidé de les classer en deux catégories : les travaux qui utilisent la traçabilité entre les artefacts comme les diagrammes d'UML ou de l'architecture et ceux qui se basent sur les dépendances extraites à partir du code source. La dernière partie de ce chapitre a été consacrée à la présentation des approches probabilistes qui ont été utilisées afin de gérer l'incertitude qui caractérise l'impact de changement. Cette revue de l'état de l'art, nous a donnés l'intuition de combiner différents aspects des travaux existants pour en tirer avantage et combler les lacunes existantes. En effet, l'inconvénient majeur des approches basées sur la traçabilité entre les différents artefacts du système reste le degré de correspondance entre les modèles qu'on utilise pour l'analyse et le code source du système.

Par ailleurs, les différentes contributions relatives à cet aspect de maintenance peuvent être classées selon leurs objectifs. En effet, les travaux présentés dans ce chapitre ont deux objectifs : la prédiction de l'impact comme illustré Tableau 1 le et l'analyse de l'impact résumée dans Tableau 2 .

Approche	Techniques	Artefact logiciel utilisé
Wong [72]	ACN	Modèle de classe UML + les données historiques
Lock [46]	Graphe de dépendances entre les besoins	Méta-modèle des besoins des utilisateurs + les données historiques
Haney [22]	Matrice de dépendances entre les modules	Données historiques
Briand [8]	Métriques de couplage	Code source + données historiques
Badri [5]	Graphe d'appel contrôlé (CCG)	Code source
Jashki [38]	Matrice de dépendences	Données historiques
Poshyvanyk [54]	Indexation sémantique latente	Code source
Tang [64]	AREL+ réseaux bayésiens	Architecture du système
Abdi [1]	Métriques de couplage+ réseaux bayésiens	Code source
Mirarab [47]	Métriques de couplage + réseaux bayésiens	Code source + données historiques
Zhou [76]	Réseaux bayésiens	Données historiques

Tableau 1: synthèse des travaux sur la prédiction de l'impact

Approche	Techniques	Artefact logiciel utilisé
RADC [58]	Règles de traçabilité	Modèles d'analyse
Hassan [23]	Système de règles	Modèle architectural et modèle structurel.
Goknil [20]	Système de règles	Méta-modèle des besoins des utilisateurs
Li **Erreur ! ource du renvoi introuvable** .	Fermeture transitive sur les graphes de dépendance	Code source
Chaumun [9][10]	Modèle d'impact	Code source
Chaumun [11]	Métriques de couplage	Code source
Kabaili [39]	Métriques de cohésion	Code source
Ren [56][57]	Comparaison sémantique entre les versions d'un logiciel	Code source
Hattori [24]	Graphe d'appel	Code source
Law [43], Orso [50], Apiwattang[2]	Traces d'exécution	Code source
Huang [25] [26]	Traces d'exécution+ graphe de dépendances	Code source

Tableau 2: synthèse des travaux sur l'analyse de l'impact

La première famille, présentée dans le Tableau 1, préconise la prédiction du changement avant qu'il soit réellement effectué sur le système. Cette première famille d'approches utilise entre autres les données historiques sur les anciens changements pour prédire les éventuelles propagations des changements. Par contre, la deuxième famille, présentée dans le Tableau 2, suppose que le changement ait été effectué et applique des techniques pour retracer ses effets relatifs sur le système. Ces techniques utilisent les dépendances extraites du code source ou d'autres artefacts logiciels et qui sont exprimées sous forme de métriques ou en ayant recours à des graphes de dépendances.

Chapitre 3
Construction des modèles d'analyse de l'impact des changements

3.1 Introduction

La maintenance est une phase importante dans le cycle de vie d'un logiciel. Diverse techniques comme l'analyse de l'impact des changements sont déployées lors de cette phase pour garantir le bon fonctionnement du système et assurer son évolution. Malgré les avancées réalisées dans ce domaine et la richesse de l'état d'art sur l'analyse de l'impact de changement, cette tâche reste toujours ardue [74]. En effet, il existe toujours une part d'incertitude difficile à gérer pour la compréhension du processus de propagation des effets d'un changement. En effet, l'analyse de l'impact des changements dépend des changements à apporter et de la manière avec laquelle les systèmes sont implémentés. Nous pensons donc qu'une représentation explicite de l'incertitude à travers une approche probabiliste permettrait de mieux opérationnaliser le raisonnement sous-jacent à la propagation des changements.

Énumérer les facteurs qui influencent l'impact de changements et analyser leurs effets reste toujours une tâche difficile. De plus, même la nature de ces facteurs est hétérogène. Cette hétérogénéité rend difficile la construction de modèles de type régression. En effet, de nombreux facteurs portent sur l'existence ou l'absence d'un type de lien entre classes. D'autres sont plutôt quantitatifs et indiquent la fréquence d'un type de relation. Pour contourner cette difficulté, nous proposons d'utiliser des modèles probabilistes (réseaux bayésiens). Ce choix est justifié par diverses raisons. D'une part, les réseaux bayésiens peuvent intégrer l'incertitude dans le raisonnement. D'autre part, ils nous permettent de combiner des données de nature hétérogène. Finalement, les réseaux bayésiens offrent une présentation intuitive en permettant la représentation graphique des relations de type cause-effet reliant les nœuds parents au nœud fils.

L'existence d'une relation d'influence de la propagation du changement ne signifie pas que la propagation est systématique. En effet, la même relation peut propager le changement dans certains cas et pas dans d'autres. Les réseaux bayésiens offrent un moyen efficace de représenter ces variations à travers les probabilités. Les probabilités peuvent être apprises à partir de données historiques et être mises à jour au fur et à mesure que le réseau est utilisé.

Nous proposons une approche probabiliste qui, suite à un changement dans une classe, prédit la probabilité d'impact des autres classes du système. Cette probabilité dépend des facteurs d'influence qui sont les liens reliant la classe changée avec le reste des classes et du type du changement à appliquer. Nous utilisons les réseaux bayésiens pour la mise en œuvre de notre approche. Ensuite nous examinons la qualité de nos prédictions en termes d'exactitude (est-ce que nous prédictions sont correctes?) et en termes de complétude (est-ce que nous avons prédit l'ensemble des classes affectées avec une grande probabilité?)

3.2 Rappel sur les réseaux bayésiens

3.2.1 Définition

Un réseau bayésien, comme illustré dans la Figure 4, est un graphe orienté acyclique noté G (V,E). V représente l'ensemble des nœuds du graphe et E représente les arcs reliant ces nœuds. Chaque nœud du réseau bayésien représente une variable aléatoire avec un nombre fini d'états. À chaque variable aléatoire correspond une table de probabilité. Cette dernière exprime la probabilité conditionnelle entre un nœud fils et ses parents. En effet, les arcs dans un réseau bayésien définissent une relation entre un nœud père (nœud de départ) et son enfant (nœud d'arrivée). Cette relation entre pères et fils se reflète dans la table de probabilité relative à chaque nœud. Ainsi la table de probabilité relative à une variable a la forme de probabilité (variable | parents) ou littéralement la probabilité d'une variable « enfant » sachant les états des variables « parents ». Par ailleurs, Naïm et al. [48] définissent les réseaux bayésiens comme étant au carrefour de la théorie des graphes et de la théorie des probabilités.

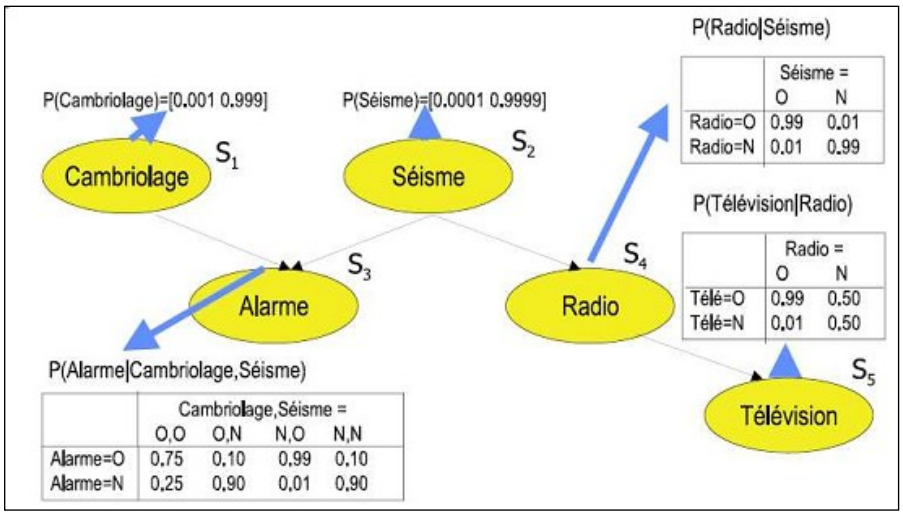

Figure 4: exemple de réseau bayésien [27]

Le calcul de ces probabilités est basé sur le théorème de Bayes:

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}.$$

Cette expression peut traduire que la probabilité de la réalisation de l'événement A sachant B est égale à la probabilité de la réalisation de ces deux événements ensemble divisée par la probabilité de la réalisation de l'événement B.

Dans le théorème de Bayes, les composantes de cette équation sont nommées comme suit :

- A : est appelée hypothèse et B est appelée une évidence.
- P (A): est appelée la probabilité à priori. Elle correspond à la probabilité inconditionnelle que l'événement A se produise avant qu'on sache l'occurrence d'un autre événement relié à A.
- P (A|B): est appelée la probabilité a posteriori. Cette la probabilité que l'hypothèse (l'événement A dans notre cas) se produise en prenant en compte que l'évidence (l'événement B) se réalise.
- P (B|A): est appelée la fonction de vraisemblance. Elle indique la probabilité que l'événement B se produise en sachant que la réalisation de l'hypothèse (l'événement A).
- P (B) : est appelée la probabilité marginale relative à l'évènement B.

Le théorème de Bayes est une dérivation des règles de la probabilité conditionnelle. Cette dérivation se réalise comme suit.

La probabilité que l'événement A se produise sachant B est calculée avec cette formule :

$$P(A \mid B) = \frac{\mathrm{P(A \cap B)}}{\mathrm{P(B)}}.$$

La probabilité que l'événement B se produise sachant A est calculée avec cette formule :

$$P(B \mid A) = \frac{\mathrm{P(A \cap B)}}{\mathrm{P(A)}}.$$

Ainsi nous pouvons déduire cette équivalence :

$$P(A \mid B) * \mathrm{P(B)} = \mathrm{P(A \cap B)} = P(B \mid A) * \mathrm{P(A)}.$$

Par conséquent, nous obtenons la formule du théorème de Bayes.

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}.$$

Ce théorème est mis en œuvre dans les réseaux bayésiens afin de calculer les tables de probabilités des nœuds. Les nœuds pères sont assimilés aux évidences et les nœuds enfants aux hypothèses. Ainsi, en appliquant ce théorème sur les réseaux bayésiens, on peut calculer les tables de probabilités des nœuds qui dépendent de la distribution de probabilité de leurs parents directs.

Par exemple, comme illustré dans la Figure 4, la table de probabilité du nœud *Télévision* dépend de son parent direct le nœud *Radio* et le nœud *alarme* dépend à la fois des valeurs des nœuds *cambriolage* et *séisme*. Par ailleurs, une modification dans le nœud *séisme* aura un effet immédiat sur le nœud *radio*. La mise à jour de la table de probabilité de ce dernier influencera la distribution du nœud *Télévision*. Ceci s'explique par le fait que l'information se propage dans les réseaux bayésiens et se transmet graduellement d'un niveau hiérarchique à un autre.

3.2.2 Fonctionnement

Les définitions précédentes montrent qu'il existe deux volets essentiels pour l'élaboration de réseaux bayésiens : (1) construction de la structure et (2) définition des probabilités associées. La construction de la structure du graphe est une étape cruciale dans la mise en place des réseaux, car identifier les variables à étudier et les structurer selon une certaine logique a une grande influence sur la qualité des prédictions.

Cependant, concevoir un réseau qui reflète le raisonnement voulu n'est pas évident et ceci est d'autant plus vrai lors de la modélisation de systèmes complexes. Pour pallier ce problème, Neil, Fenton et Nielsen [49] proposent l'emploi des idiomes. Les idiomes sont des blocs qui représentent une bibliothèque de patrons servant à l'élaboration de la structure des réseaux. Une fois les nœuds intervenants sont définis, l'utilisation des idiomes aide à identifier la sémantique et la structure du réseau. Les auteurs ont défini cinq types d'idiomes qui représentent les différents types de liens pouvant reliés les nœuds d'un réseau bayésien:

- Idiome de définition/synthèse: c'est la modélisation d'une relation d'incertitude lors d'une définition d'un terme. Par exemple, on peut avoir une relation de définition entre les nœuds pères *masse* et *gravité* et le nœud fils *poids*.
- Idiome de cause/conséquence : c'est l'utilisation typique des réseaux bayésiens comme l'exemple de l'alarme qui sonne soit à cause du cambriolage soit à cause d'un séisme.
- Idiome de mesure : cet idiome modélise l'incertitude de la précision d'un instrument de mesure. Afin d'améliorer la qualité les observations collectées, on peut introduire une notion de risque par rapport à l'exactitude des données. Ainsi, le nœud fils représente une évaluation d'un des parents en fonction d'autres nœuds pères qui sont des estimateurs d'exactitude.
- Idiome d'induction: cet idiome modélise l'incertitude liée au raisonnement inductif basé sur des entités similaires ou échangeables. Par exemple, on peut induire l'estimation de la sécurité d'un véhicule en fonction des historiques des accidents et de la similarité entre la voiture en question et celles intervenant dans les accidents passés.
- Idiome de réconciliation : cet idiome modélise la réconciliation de résultats liés à deux idiomes différents. En effet, les données collectées peuvent parvenir de différentes sources ou traitées par des idiomes différents. L'objectif de cet idiome est de joindre des parties du réseau différentes. On peut par exemple combiner deux sous réseaux pour évaluer le risque de Tsunami en combinant un sous réseau relatif à l'évaluation des données des séismes avec l'idiome conçu pour l'analyse des risques de tornades.

Ces idiomes peuvent être réutilisés afin d'accélérer le processus de conception des structures des réseaux bayésiens.

Par ailleurs, il existe des techniques qui identifient la structure des réseaux en se basant sur un ensemble d'observations. Ce processus est appelé l'apprentissage de la structure. L'objectif de cet apprentissage est de trouver le graphe qui exprime le mieux les relations de causalité entre les variables. Par ailleurs, l'énumération de toutes les combinaisons possibles peut engendrer une explosion combinatoire.

Ainsi, deux grandes familles d'approches sont utilisées pour contourner cette problématique. La première a pour objectif de trouver un réseau qui maximise un score reflétant le degré de corrélation entre les variables et la structure proposée. On peut citer les algorithmes de l'arbre de poids maximal [14], Structural-EM [18] et K2 [19] comme exemples de cette catégorie d'approche. La deuxième famille utilise les tests d'indépendance conditionnelle qui consistent en des tests statistiques pour déterminer des contraintes d'indépendances entre les variables. Ceci se traduit par la suppression des arcs reliant des variables indépendantes dans le réseau bayésien. Des algorithmes comme PC [63] et BN-PC [12] sont utilisés pour la mise en œuvre de cette catégorie d'apprentissage de structure.

Une fois que les variables (nœuds) sont représentées et que les relations entre elles sont identifiées, la deuxième étape est la génération des tables de probabilités relatives à chaque nœud. Les tables de probabilités peuvent être obtenues à l'aide des avis des experts et/ou par apprentissage réalisé sur des données historiques. Tout comme pour l'apprentissage de la structure, il existe plusieurs algorithmes qui réalisent l'apprentissage des paramètres. On peut identifier deux grandes familles de méthodes. La première inclut les méthodes qui supposent l'existence de données suffisantes pour faire l'apprentissage. Dans ce genre d'approche, on utilise des techniques comme le maximum de vraisemblance [62] qui représente la fréquence d'apparition de l'événement dans la base d'exemples utilisée pour l'apprentissage. Les méthodes de la seconde famille supposent que les données sont incomplètes. En effet, il se peut que certaines variables ne soient observées que partiellement ou même jamais. Parmi les travaux dans cette famille, on peut citer le travail de Dempster, Laird et Rubin [15]. Dans cet article, les auteurs proposent

l'algorithme EM (Expectation-Maximisation) pour réaliser l'apprentissage des paramètres avec des données incomplètes.

3.2.3 Mise en œuvre des réseaux bayésiens : l'inférence

Une fois que la structure du réseau est mise en place et que les tables de probabilité sont définies, le réseau bayésien est utilisé pour calculer des probabilités. Ce procédé est appelé l'inférence bayésienne. L'inférence bayésienne est définie par Naim et al. [48] comme « le processus de propager une ou plusieurs informations certaines au sein d'un réseau pour en déduire comment sont modifiées les croyances concernant les autres nœuds ». En d'autres termes, l'inférence sert à calculer la probabilité d'une hypothèse suite à l'observation des évidences. Les évidences correspondent aux noeuds d'entrée et les hypothèses sont les différents états des nœuds de sortie du réseau. L'injection des probabilités des nœuds d'entrée va modifier récursivement les probabilités des nœuds enfants jusqu'aux nœuds de sortie. Le calcul des probabilités utilise à la fois les tables de probabilités et le théorème de Bayes.

3.3 Construction de la structure des réseaux

Dans notre approche, nous proposons une collection d'idiomes qui vont correspondre aux changements que nous comptons apporter au code source. Les changements que nous considérons sont de différentes granularités. Ils touchent les classes, les méthodes et les attributs. Notre revue de littérature nous a indiqué que la majorité des travaux se focalisent uniquement sur un type de changement ou un niveau granularité en particulier. À travers ce mémoire, nous traitons différentes granularités pour pouvoir traiter un nombre maximal des changements que les programmeurs comptent apporter aux systèmes. Cependant, avoir la possibilité de gérer différents types de changements nécessite de trouver les facteurs qui influencent la propagation de chaque type de changement. Afin de déterminer les nœuds représentant ces facteurs, on s'est inspiré du modèle de Chaumun et al. [10]. Ce travail avance l'idée que l'impact de changement peut se justifier par l'existence de certains types de relations entre les classes. Ces relations sont de même type que celles qu'on trouve dans les diagrammes de classe UML tels que l'association, l'agrégation et l'héritage. De plus, les auteurs définissent un lien interne pour exprimer un impact local au niveau de la classe elle-même. Les auteurs vérifient aussi

l'existence ou non d'une invocation entre les classes. Dans notre travail, nous avons omis le lien de l'impact local, car notre approche sert à déterminer la probabilité que d'autres classes soient affectées suite à un changement donné. La classe où le changement est effectué doit toujours être testée, car elle serait évidemment affectée.

Par ailleurs, le lien d'invocation dans notre approche n'est pas booléen. En effet, nous pensons que si on change une classe alors une classe avec un nombre d'invocations élevé a plus de chance d'être affectée qu'une autre classe dont le nombre d'invocations avec la classe changée est faible. En plus, le lien d'héritage que nous utilisons s'intéresse aux changements effectués dans la classe « mère » et qui peuvent affecter ses classes « filles ».

Le travail défini par Chaumun propose la détection des liens en montrant des exemples de situations dans le code source qui peuvent correspondre aux liens utilisés dans l'approche. Par exemple, si une classe C1 utilise un attribut de type C2 alors C1 et C2 sont reliées par un lien d'association. L'inconvénient majeur dans cette démarche est que l'extraction des liens d'association et d'agrégation n'est pas clairement définie. En effet, il manque une équivalence formelle entre les structures d'un programme et ces liens abstraits.

Pour pallier ce problème, nous utilisons une technique proposée par Guéhéneuc et Albin-Amiot [21] pour la rétro-ingénierie des diagrammes de classe. Dans ce travail, les auteurs commencent par définir les termes « association », « agrégation » et « composition » au niveau du code source et du diagramme de classes. Ensuite, ils présentent quatre propriétés afin de formaliser les définitions présentées: l'exclusion, l'invocation, la durée de vie et la cardinalité.

Le rôle de ces propriétés, écrites sous forme de conjonction, est d'exprimer les relations entre les classes. Par exemple, pour illustrer la propriété durée de vie, on considère deux classes A et B. Si ces deux classes sont reliées par une agrégation, alors au moment de créer un objet de type A, on crée un objet de type B. Pour la même propriété, mais avec une relation de composition, les objets A et B doivent être créés et détruits au même moment. Ces propriétés sont utilisées dans un algorithme afin de détecter les liens du diagramme UML existants dans le code source. L'approche proposée a été testée sur des systèmes tels que JHOTDRAW et ArgoUML dont les auteurs avaient le diagramme

UML au préalable. L'approche proposée arrive à identifier 96% des relations du diagramme UML et 75% des relations identifiées correspondent à ce qu'existe dans le diagramme. Ces résultats satisfaisants nous ont incités à opter pour cette technique afin de déduire les liens responsables de la propagation des changements. Cependant, les outils utilisés pour cette approche ne nous permettaient pas de connaître le nombre de liens entre les classes du système.

Nous avons donc développé un outil dans le cadre de développement Soot [66]. Soot a été conçu pour l'analyse et l'optimisation du code Java. Il nous a permis d'obtenir les informations relatives au nombre d'invocation entre les classes des systèmes. Par conséquent, nous avons implémenté un module qui prend comme point d'entrée le projet à étudier. Le résultat produit est une matrice M dont les lignes et les colonnes sont les classes du système. Chaque élément $M_{(i,j)}$ correspond à un quadruplet (H,G,A,I) :

- H=T s'il existe un lien d'héritage entre i et j
  sinon H=F.
- G=T s'il existe un lien d'agrégation entre i et j
  sinon G=F.
- A=T s'il existe un lien d'association entre i et j
  sinon A=F.
- I correspond au nombre d'invocations entre la classe i et j.

	org.apache.xerces.dom.DocumentImpl	org.apache.xerces.dom.ParentNode	org.apache.xerces.dom.AttributeMap	org.apache.xerces.dom.NamedNodeMapImpl	org.apache.xerces.dom.DocumentTypeImpl	org.apache.xerces.dom.NodeImpl	org.apache.xerces.dom.DeepNodeListImpl	org.apache.xerces.dom.ElementDefinitionImpl
org.apache.html.dom.HTMLDocumentImpl	F,F,F,4	F,F,F,1	F,F,F,1	F,F,F,1	F,F,F,0	F,F,F,0	F,F,F,0	F,F,F,0
org.apache.xerces.dom.ElementImpl	F,F,A,23	F,F,F,13	F,G,F,5	F,F,F,6	F,F,F,0	F,F,F,6	F,F,F,0	F,F,F,0
org.w3c.dom.html.HTMLElement	F,F,F,0	F,F,F,0	F,F,F,0	F,F,F,0	F,F,F,0	F,F,F,0	F,F,F,0	F,F,F,0
org.w3c.dom.html.HTMLFormElement	F,F,F,0	F,F,F,0	F,F,F,0	F,F,F,0	F,F,F,0	F,F,F,0	F,F,F,0	F,F,F,0
org.apache.xerces.dom.ChildNode	H,F,F,1	H,F,A,16	F,F,F,1	F,F,F,1	H,F,F,0	F,F,F,4	F,F,F,0	H,F,F,0
org.apache.xerces.dom.DocumentImpl	F,F,F,78	F,G,A,21	F,F,F,7	F,F,F,7	F,G,F,0	F,G,F,10	F,F,F,0	F,G,F,0
org.apache.xerces.dom.ParentNode	H,F,F,9	F,F,F,25	F,F,F,9	F,F,F,6	H,F,F,7	F,F,F,13	F,F,F,0	H,F,F,3
org.apache.xerces.dom.AttributeMap	F,F,F,2	F,F,F,0	F,F,F,3	F,F,F,1	F,F,F,3	F,F,F,0	F,F,F,0	F,F,F,1
org.apache.xerces.dom.NamedNodeMapImpl	F,F,F,8	F,F,F,0	H,G,F,9	F,F,F,10	F,F,A,9	F,F,F,2	F,F,F,0	F,F,A,2
org.apache.xerces.dom.DocumentTypeImpl	F,F,A,27	F,F,F,12	F,F,F,5	F,F,F,6	F,F,F,1	F,F,F,3	F,F,F,0	F,F,F,0
org.apache.xerces.dom.NodeImpl	H,F,F,26	H,F,F,53	F,F,F,30	F,G,A,12	H,F,F,12	F,F,A,41	F,G,A,0	H,F,F,3

Figure 5: exemple de matrice des liens (Xerces 1.3.1)

Une fois les données requises obtenues, l'étape suivante consiste à choisir et à regrouper les liens adéquats pour chaque type de changement. En effet, chaque type de changement se propage à travers un ou plusieurs liens du quadruplet (H,G,A,I). Cette étape est d'une grande importance vu que la structure des réseaux influe considérablement sur les valeurs prédites.

Le modèle de Chaumun [10] était notre point de départ. Dans ce travail, les classes affectées sont identifiées selon leurs liens avec la classe à changer. Ce choix a donné des résultats prometteurs, mais on a remarqué que certaines combinaisons doivent être supprimées ou modifiées afin de mieux cerner la propagation de changement. En effet, ce travail a été élaboré pour des programmes codés en C++ et il existe des leins comme « Friendship » qu'on ne trouve pas dans le langage Java. Ce constat vient des tests manuels que nous avons effectués sur des systèmes de différentes tailles dont les auteurs fournissent les diagrammes de classes comme JUnit [35], JHotDraw [34] ou une application de gestion bancaire [36]. Ainsi, nous injectons des changements et nous compilons le système pour

avoir la liste des classes affectées (celles qui ne compilent pas). Par la suite, nous prenons en considération les liens entre la classe changée et celle qui est affectée.

Nos constatations nous ont menés à faire des modifications sur le modèle défini. Par exemple, pour le changement qui consiste à supprimer un attribut, le modèle de Chaumun propose comme classes affectées celles qui sont reliées avec la classe changée par un lien d'association. Nos résultats empiriques ont montré que certaines classes reliées par des liens d'héritage et d'invocations l'étaient aussi. Nous avons donc rajouté les liens d'héritage et d'invocations dans notre modèle. En plus, dans le modèle de Chaumum, l'ajout de variables et des méthodes n'a aucun effet sur le système. Ceci n'est pas toujours vrai. En effet, si l'on rajoute une méthode à une classe abstraite, alors les classes qui l'implémentent doivent prendre en considération ce type de changement. Finalement, nous avons obtenu les structures des réseaux, illustrés dans les figures [6;12] que nous utilisons et que nous testons à travers ce mémoire. Ces réseaux peuvent être classés en trois catégories selon la granularité choisie.

La première catégorie de réseau est celle qui s'intéresse aux changements relatifs aux classes du système. La Figure 6 illustre le réseau utilisé pour la détection des classes affectées suite au changement de signature d'une classe. Quand on modifie une classe, les classes qui héritent de cette classe ou qui sont en association avec elle ou qui l'appellent doivent être testées. Ainsi le réseau est composé de trois liens comme entrée qui servent à déterminer l'impact de changement. Le deuxième changement relatif aux classes est la surpression. Le réseau illustré dans la Figure 7 illustre les différents liens qui sont responsables de la propagation de ce type de changement. En effet, la suppression d'une classe va entraîner un impact sur les classes avec lesquelles elle était en relation et ceci indépendamment du type de lien.

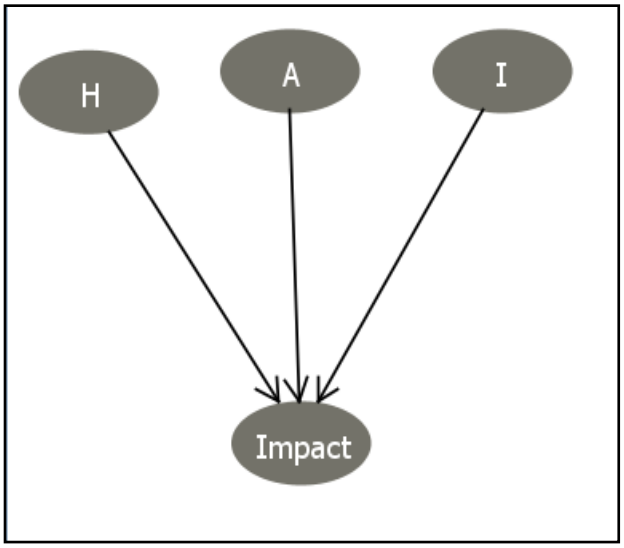

Figure 6: réseau bayésien pour le changement des classes

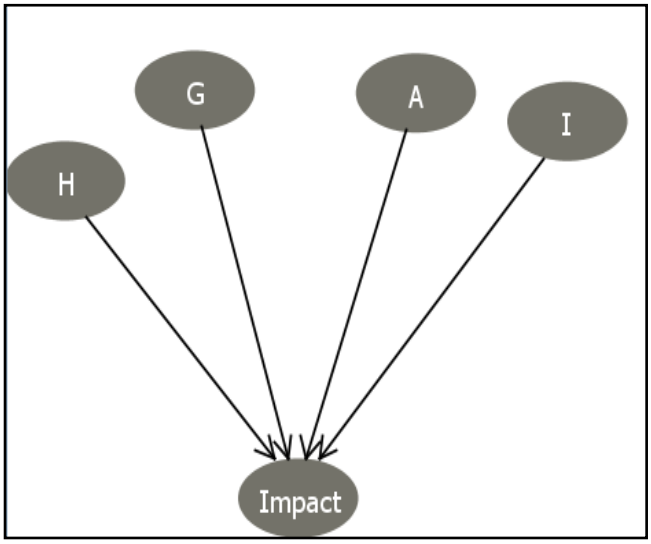

Figure 7: réseau bayésiens pour la suppression des classes

La deuxième catégorie de réseau est celle qui s'intéresse aux changements relatifs aux méthodes du système. Nous pensons que la combinaison de liens, illustrée dans la Figure 8, soit la meilleure pour identifier l'impact de l'ajout d'une méthode. En effet, ce type de changement se propage généralement à travers les liens d'héritage, d'association et d'invocation.

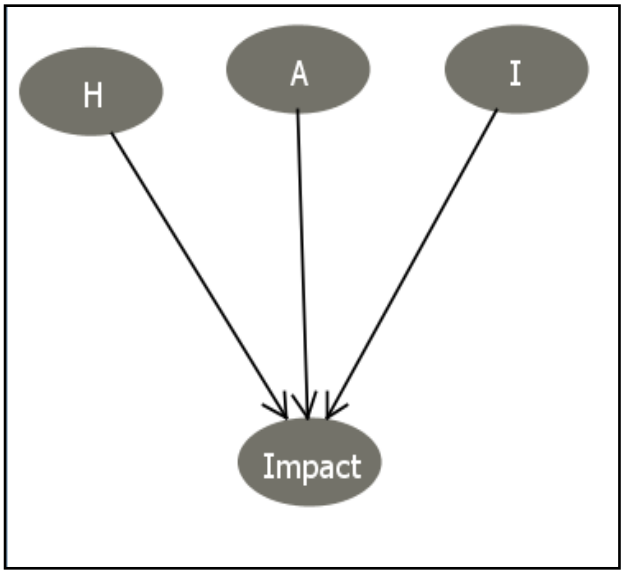

Figure 8: réseau bayésien pour l'ajout de méthodes

Un changement de méthode pourrait affecter les classes qui utilisent cette méthode. Ainsi nous avons choisi d'inclure l'ensemble des liens dans le réseau utilisé pour prédire les classes affectées comme le montre la Figure 9. Nous utilisons le réseau bayésien illustré dans la Figure 10 pour le changement qui consiste la suppression de méthode. En effet, en appliquant ce changement, les classes qui appellent cette méthode (I) ou qui héritent cette méthode doivent être analysées. De plus, la suppression d'une méthode peut briser un lien d'agrégation entre deux classes.

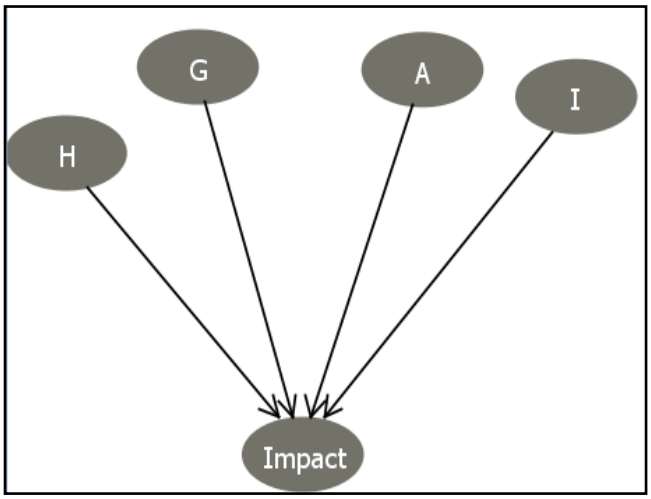

Figure 9: réseau bayésien pour le changement de méthodes

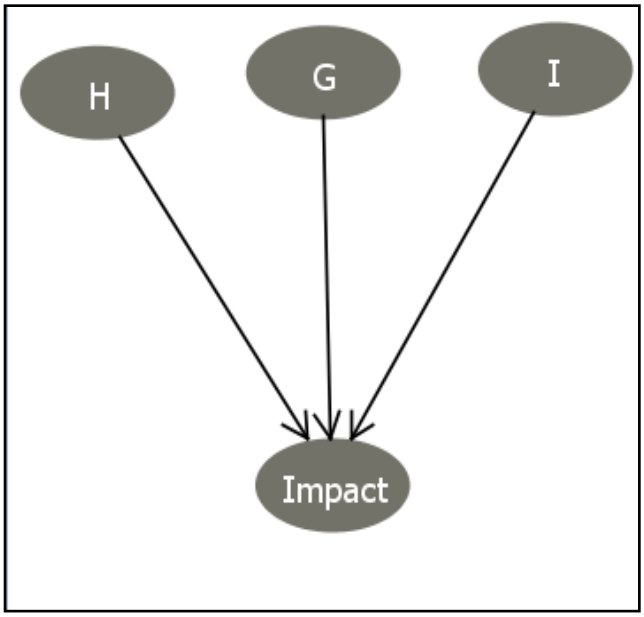

Figure 10: réseau bayésien pour la suppression de méthodes

La dernière granularité à laquelle nous nous intéressons est les attributs. Les modifications d'attributs sont rares dans la pratique (surtout que les modifications des attributs dans notre modèle de changement excluent les changements de nom). Par conséquent, nous étions incapables de rassembler des données suffisantes pour ce type de changement afin d'apprendre et de tester notre modèle de prédiction.

Dans ce projet, nous présentons deux réseaux qui peuvent être utilisés afin de prédire les classes affectées suite à un ajout ou la suppression d'un attribut. La Figure 11 indique les liens que nous pensons responsables de la propagation d'un impact dans les autres classes du système suite à un ajout d'attribut. En effet, les classes qui sont susceptibles d'être affectées sont celles qui sont reliées par un lien d'association et d'invocation.

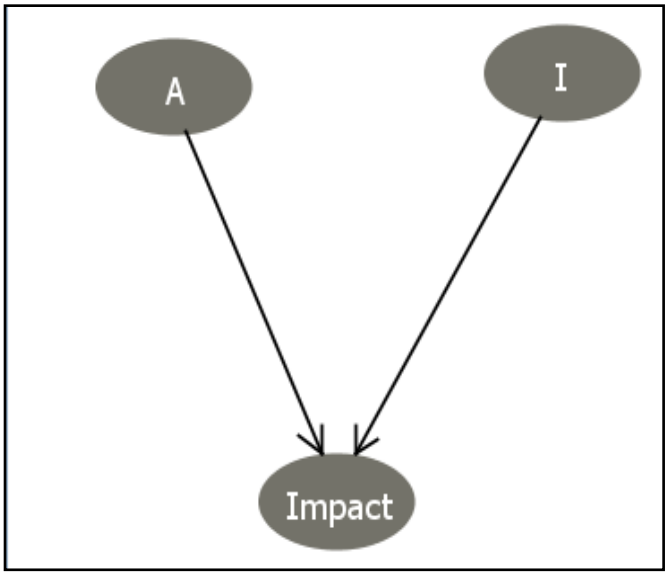

Figure 11: réseau bayésien pour l'ajout d'attribut

La suppression d'un attribut de classe peut supprimer un lien d'association. De plus, les classes qui héritent de la classe changée risquent d'être affectées. Ainsi, nous avons choisi d'inclure ces liens comme nœuds d'entrée dans notre réseau illustré dans la Figure 12.

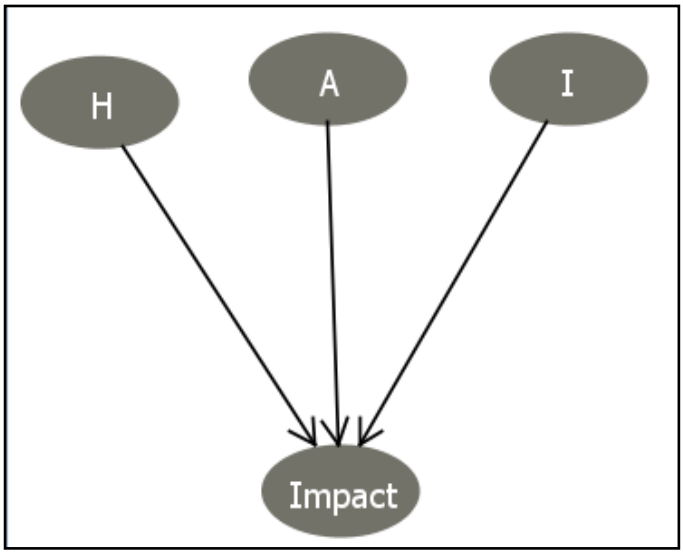

Figure 12: réseau bayésien pour la suppression d'attribut

3.4 Paramétrage des réseaux bayésiens

L'étape qui suit la création des structures est le calibrage des tables de probabilité des nœuds. Les réseaux bayésiens peuvent intégrer l'avis des experts pour la définition des tables de probabilités. Plusieurs travaux relatifs à l'analyse de l'impact de changement choisissent cette option pour calibrer les tables de probabilités. Cependant, les experts peuvent ne pas être disponibles dans tous les projets et ils représentent un coût qui s'ajoutera au coût déjà élevé de la maintenance. Par conséquent, nous avons opté pour un apprentissage des paramètres à partir de données réelles. En effet, la disponibilité des entrepôts de données et de logiciels de gestion de versions procure une importante quantité de données. Dans ce projet, nous profitons de ces données pour collecter des informations relatives à l'analyse de l'impact de changement. Le processus d'apprentissage que nous utilisons est illustré dans la Figure 13.

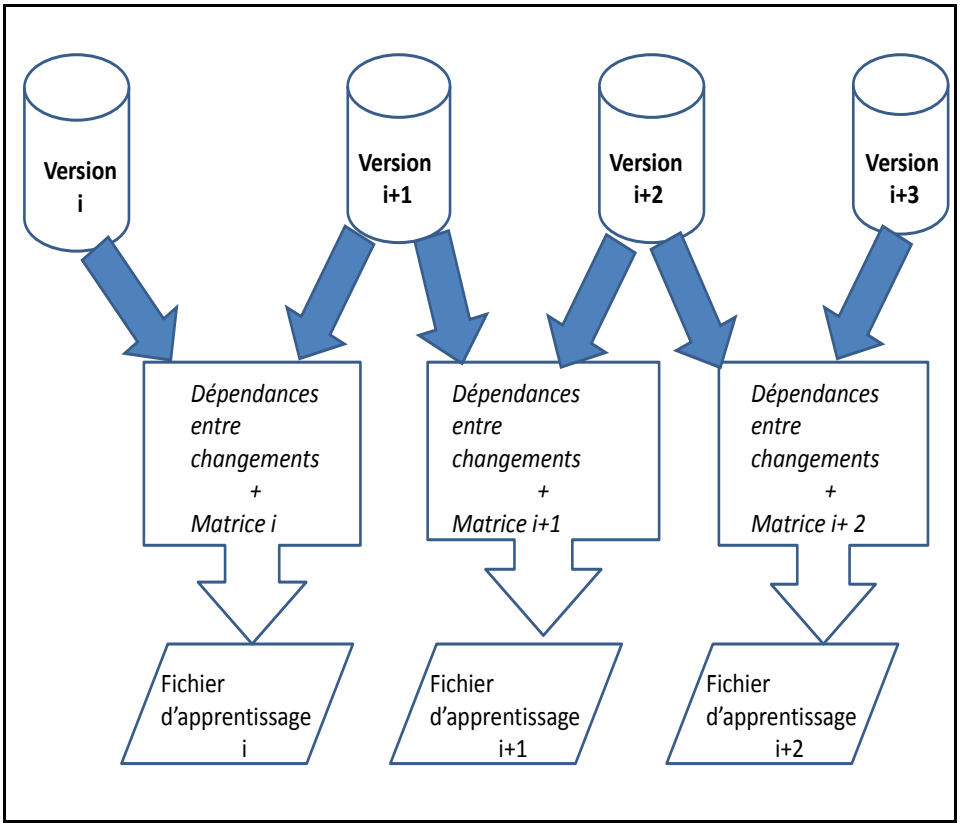

Figure 13: processus d'apprentissage des réseaux

Le processus commence par utiliser deux versions successives d'un système. Ces versions servent de point d'entrée pour Chianti [57]. Nous utilisons cet outil afin d'avoir

l'ensemble des changements atomiques qui diffèrent entre deux versions successives. De plus, Chianti offre la possibilité de trouver les dépendances entre ces changements. Ainsi, il est possible de déterminer si un changement effectué dans une classe X a nécessité d'effectuer au préalable d'autres changements dans d'autres classes du système. Ces informations, par rapport aux changements, sont stockées dans un graphe où les nœuds représentent les changements et les arcs les dépendances entre les changements. Par la suite, nous utilisons l'algorithme de Warshall [68] pour construire la fermeture transitive du changement comme illustré dans la Figure 14.

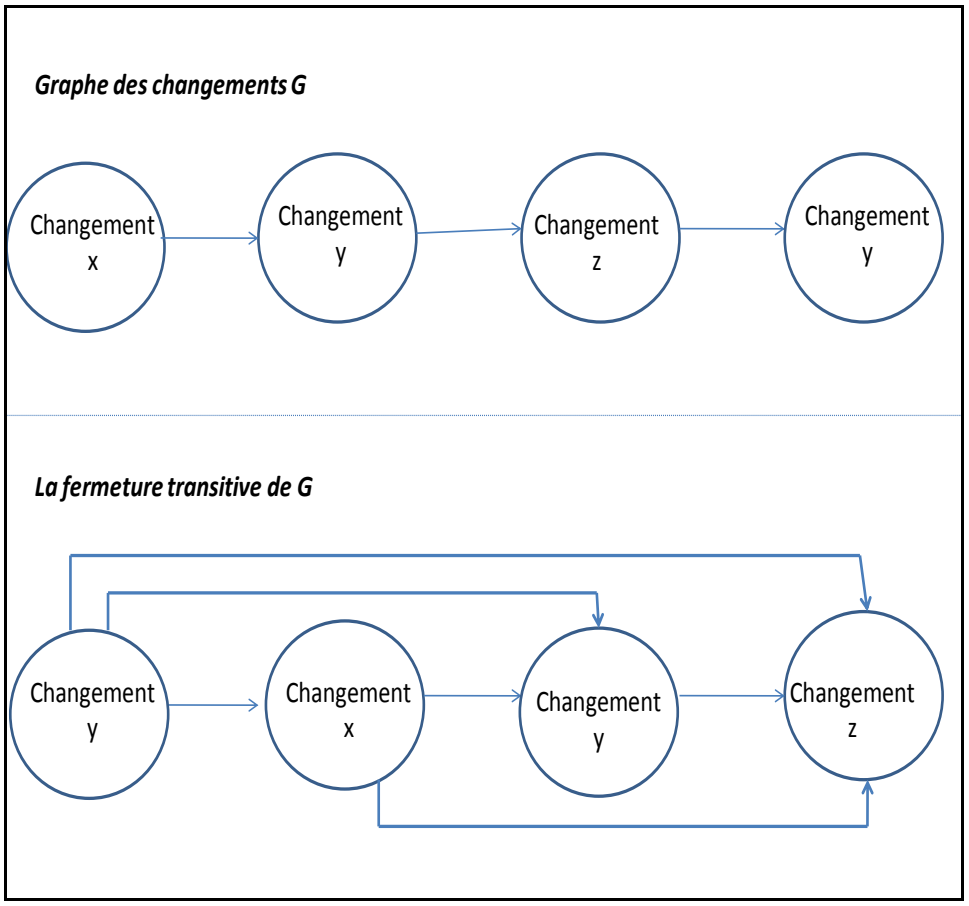

Figure 14: graphes des changements et fermeture transitive

La fermeture transitive d'un graphe orienté G (V,E) avec V représentant l'ensemble des nœuds et E l'ensemble des arcs est un graphe $G^{+}(V,E^{+})$. Pour tout x et y appartenant à

l'ensemble V il existe un arc (x,y) dans E^+ s'il existe un chemin commençant par x et passant par y dans le graphe G. L'intérêt d'appliquer cette fermeture transitive est de déduire tous les changements qui dépendent directement ou indirectement du changement que les programmeurs ont réalisé. Ainsi, notre approche prend en considération le « ripple effect ».

Par la suite, nous nous intéressons aux liens entre la classe où le changement est effectué et la classe où l'on doit faire des changements pour garder le bon fonctionnement du système suite au changement initial. Cette information existe dans la matrice des liens que nous avons déjà extraite (Voir section 3.3). Cependant, une transformation doit être réalisée au niveau de l'attribut qui indique le nombre d'invocations entre les classes. En effet, cet attribut quantitatif est calculé pour chaque couple de classes et par conséquent il peut avoir une infinité de valeurs. Ceci pose un problème lors de l'apprentissage des paramètres des réseaux bayésiens qui nécessite que chaque variable ait une table de probabilité avec un nombre d'états fini.

Pour contourner ce problème, il est nécessaire de définir un nombre fini de classes d'invocations. Une classe d'invocation représente un intervalle regroupant un ensemble de variables d'invocation. Ainsi, chaque valeur d'invocation va être attribuée à une classe. Ces classes n'étaient pas connues d'avance, nous avons procédé à un regroupement « clustering » afin de déduire les différentes classes d'invocation. Pour réaliser cette étape, nous avons collecté un ensemble de valeurs d'invocations provenant de divers systèmes auquel nous avons appliqué un algorithme de « clustering ». Comme l'affirment Xu et Wunch [73], il n'existe pas d'algorithme de « clustering » idéal. En effet l'efficacité d'un algorithme de « clustering » dépend du contexte dans lequel il est utilisé.

Dans notre mémoire, nous appliquons l'algorithme X-Means. Cet algorithme est l'extension de l'algorithme K-Means qui représente une des techniques de classification non supervisée. L'algorithme K-Means consiste à décomposer les valeurs à traiter en K sous groupes. Ensuite, l'algorithme calcule le centroïde de chacun de sous groupes. La dernière étape consiste à placer les valeurs dans le sous-groupe qui offre la distance minimale entre la valeur traitée et le centroïde du sous-groupe. Selon [73], K-Means est un algorithme qui peut être utilisé pour résoudre plusieurs problèmes pratiques. K-Means est un algorithme relativement simple à implémenter. Ceci explique le fait qu'il est un des

algorithmes les plus populaires en recherche scientifique et en applications industrielles comme le mentionne Berkhin [6]. De plus, sa complexité temporelle n'est que de l'ordre de O (NKd) avec N le nombre de points de données, K le nombre de groupes et d la dimension de l'espace utilisé pour présenter les données. Vu que K et d sont généralement plus petits que N, K-Means a une complexité temporelle raisonnable et donc un temps d'exécution acceptable (qui est inférieur à $O(N^2)$). Ceci lui permet d'être utilisé pour traiter un grand nombre de données. L'inconvénient majeur de K-Means est le choix du nombre de sous-groupes qui doit être fourni par l'utilisateur dans la première étape. Pour y remédier, X-Means offre plus de liberté, car il demande un intervalle de nombre de clusters que l'utilisateur pense être raisonnable. Ensuite, l'algorithme commence par la borne inférieure comme nombre de clusters et calcule un score. Ce score reflète si les centroïdes couvrent l'ensemble des valeurs à traiter. L'application de X-Means sur le nombre des invocations a fait ressortir trois classes: invocation faible, moyenne ou forte.

Ainsi, nous obtenons des fichiers d'apprentissage comme celui présenté dans la Figure 15. Chaque fichier d'apprentissage est relatif à un changement donné et contient deux types d'informations. Premièrement, l'entête comporte la structure du réseau et les valeurs possibles pour chaque nœud. Deuxièmement, chaque cas d'apprentissage (ligne de fichier) indique les informations relatives aux types des liens qui sont l'héritage, l'association et l'agrégation et le type d'invocation reliant les deux classes Java du système.

```
@relation CM
@attribute H{H,F}
@attribute G{G,F}
@attribute A{A,F}
@attribute Invoc{Strong,Medium,Weak}
@attribute Impact {True,False}
@data
F,G,F,Weak,True
F,G,F,Weak,True
F,F,F,Weak,False
F,F,F,Weak,False
F,F,F,Weak,False
F,F,F,Weak,False
F,F,F,Weak,False
F,G,A,Weak,True
F,G,F,Weak,True
F,F,F,Weak,False
F,G,A,Weak,True
F,G,A,Weak,False
F,F,F,Weak,False
F,F,F,Weak,False
F,G,F,Medium,True
F,F,F,Medium,False
F,G,A,Medium,True
F,G,A,Medium,True
F,G,A,Medium,True
H,F,F,Medium,True
F,F,F,Medium,True
F,G,A,Strong,True
F,G,A,Strong,True
F,G,A,Strong,True
F,G,A,Strong,True
F,G,A,Strong,True
F,G,A,Strong,True
F,G,A,Strong,True
F,G,A,Strong,True
F,G,A,Strong,True
F,F,F,Weak,False
F,F,F,Weak,False
F,F,F,Weak,False
```

Figure 15: exemple de fichier d'apprentissage

Pour garantir un apprentissage de qualité, en plus des cas positifs, on introduit dans les fichiers d'apprentissage des occurrences des classes qui ne sont pas affectées par un changement (cas négatifs). En effet, si un changement x dans une classe A affecte une classe B, on introduit cette information pour déduire la propagation du changement. De plus, par ce même changement x, on choisit au hasard d'autres classes qui ne sont pas affectées. Par conséquent, le système aura deux types d'informations : les combinaisons de liens qui propagent le changement et les combinaisons des liens qui ne le font pas.

3.5 Conclusion

À travers ce chapitre, nous présentons notre approche afin de prédire l'impact des changements en utilisant les réseaux bayésiens. Cette approche peut être résumée en trois étapes. La première étape est la génération des informations que nous pensons pertinentes pour la propagation des changements. Ces informations sont extraites du code source et sont stockées sous forme de matrice. Ainsi, chaque cellule de la matrice indique les liens entre deux classes du système. Ensuite, nous proposons aux mainteneurs un réseau bayésien dépendamment du changement qu'ils comptent effectuer. Les réseaux bayésiens que nous utilisons ont été entraînés auparavant afin de prédire la probabilité de propagation des changements. Cet apprentissage est réalisé avec des données réelles. Dans le chapitre suivant, nous allons tester l'efficacité des structures et de l'apprentissage réalisé en appliquant des scénarios de changement sur différents systèmes.

Chapitre 4.

Mise en œuvre et évaluation

4.1 Introduction

Dans ce chapitre, nous présentons la mise en œuvre et l'évaluation de notre approche. La mise en œuvre consiste en un processus qui commence par la détection des liens responsables de la propagation des changements. Ensuite, ces dépendances servent pour exécuter les réseaux bayésiens définis dans le chapitre précédent afin de prédire les probabilités d'impact pour les classes d'un système suite à un changement.

Par la suite, nous présentons les résultats obtenus pour deux scénarios de prédiction. Le premier considère que des versions antérieures sont disponibles pour le système. Ces versions servent à réaliser l'apprentissage comme le montre la Figure 13 du chapitre 3. Le second scénario suppose l'absence de telles données. L'apprentissage se réalise avec des données collectées sur d'autres systèmes. Les résultats de l'approche que nous proposons sont bons pour les deux scénarios.

4.2 Mise en œuvre

Afin de déterminer les classes affectées suite à un changement donné, nous procédons en trois étapes comme le montre la Figure 16.

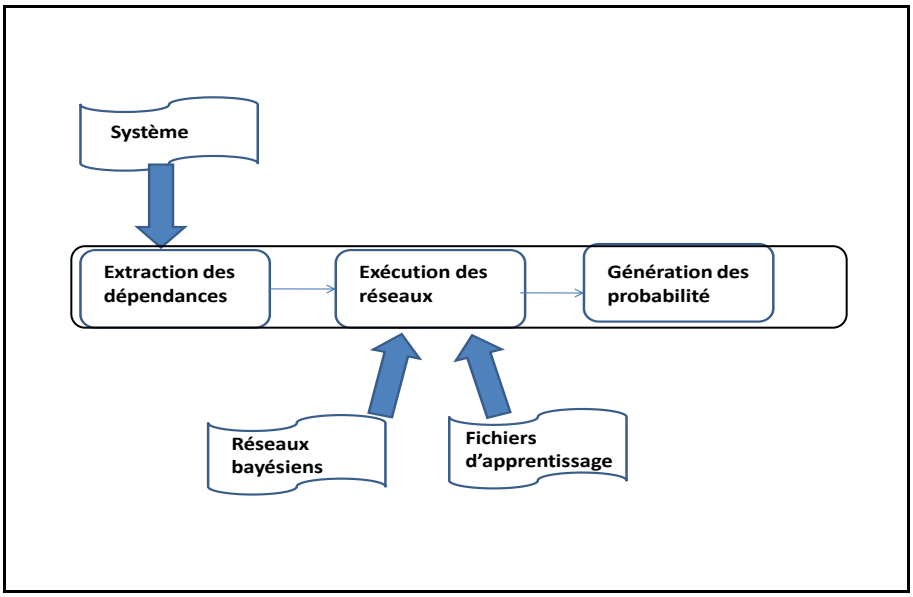

Figure 16: mise en œuvre des réseaux bayésiens

La première étape consiste à extraire les dépendances entre les classes qui peuvent être responsables de la propagation du changement. Cette étape, décrite dans le chapitre précédent, est réalisée par rétro-ingénierie à l'aide d'un outil implémenté à partir du cadre de travail Soot [66]. La deuxième étape consiste à choisir le réseau qui peut prédire la propagation en fonction du changement que les programmeurs envisagent d'apporter au système. Ces réseaux sont au préalable entraînés avec des fichiers d'apprentissage. La dernière étape consiste à appliquer l'inférence bayésienne.

Au moment d'appliquer l'inférence pour une classe donnée, on extrait les dépendances de cette classe avec les autres classes du système. Les liens d'héritage, d'agrégation et d'association sont de types booléens et leurs tables de probabilité peuvent avoir une probabilité de 0,999 si le lien existe et 0,001 s'il n'y a pas de lien. Ce choix est par défaut dans la majorité des outils qui implémentent les réseaux bayésiens. Par contre, le dernier lien, qui est le nombre d'invocations est quantitatif. Afin de réaliser l'apprentissage (section 3.4), nous optons pour l'application d'un algorithme de regroupement « clustering » classique pour attribuer à chaque valeur un type. Ce type peut être : faible, moyen ou fort. Cependant, pour une nouvelle occurrence dont on veut inférer la probabilité, nous utilisons la logique floue pour réaliser le groupement.

La logique floue a été introduite par Zadeh [75] afin de fournir un cadre conceptuel efficace pour traiter le problème de la représentation des connaissances dans un

environnement d'incertitude et d'imprécision. La logique floue est une extension de la théorie des ensembles classique. Son atout majeur est l'adaptation au raisonnement humain en laissant une place entre la certitude du vrai et la certitude du faux. En effet, dans certaines conditions il est plus judicieux de nuancer nos jugements.

Une des applications de la logique floue est le groupement flou. Zadeh définit un ensemble flou A d'un univers R^n par la fonction $\mu A: R^n \rightarrow [0, 1]$. La fonction $\mu A (x)$ est appelée la fonction d'appartenance. Elle associe à chaque élément son degré d'appartenance à une des classes définies par ce groupement. Les classes qui caractérisent un groupement sont aussi nommées les grappes. Comme le montre la fonction, la valeur d'un état flou est comprise entre 0 et 1.

Par exemple, supposons qu'on veut attribuer une description de la taille d'une personne qui peut être petite, moyenne, ou grande. Avec la théorie des ensembles classique, supposons que l'intervalle du petit est [0,150 [, du moyen [150, 175[et du grand [175, $+\infty$]. Si une personne est de taille 149 alors elle sera considérée de petite taille comme une personne dont la taille est 75 bien que ce soit la moitié de la taille. Par contre, la logique floue nous permet de dire que cette personne est à la fois de petite et moyenne taille avec des degrés de vérité divers.

Malgré que la définition de la logique floue soit différente de la théorie des probabilités, les travaux de Dubois et Prade [16] ont montré qu'on peut assimiler la fonction d'appartenance (logique floue) à une fonction de probabilité sous certaines conditions. En effet, l'algorithme de partitionnement flou doit préserver le fait que la somme des degrés d'appartenance dans toutes les grappes est égale à 1. Le processus de partitionnement flou généralise les méthodes de regroupement par grappes en permettant à une valeur d'être partiellement classée dans une ou plusieurs grappes à la fois. En effet, l'adhésion ou l'appartenance de cette valeur est distribuée dans toutes les grappes. Si ces conditions sont respectées, alors l'inférence bayésienne (mise à jour des tables de probabilités) peut être interprétée en termes d'observations floues.

En général, le nombre de groupes ou grappes à choisir n'est pas connu au préalable. Le groupement peut se réaliser en utilisant le degré de similitude et de différence entre les observations afin de déterminer le nombre de groupes. Ce dernier peut aussi être déterminé empiriquement en utilisant des techniques comme le coefficient de Dunn [17]. Dunn s'est

intéressé à la recherche des classes compactes et séparables « Compact Well Separated Clusters » dans un ensemble de données. Ce coefficient varie de 1/k (classification complètement floue) à 1 (équivalent à un groupement classique). Ce résultat, comme l'affirme Trauwaert [65], permet de déduire que le nombre de grappes qui optimise le groupement est celui maximisant ce coefficient.

Afin de mettre en œuvre la logique floue pour dériver les probabilités pour le nœud correspondant aux invocations, on commence par collecter les valeurs d'invocations entre les classes de différents systèmes. Le choix de diversifier les systèmes est justifié par notre volonté d'avoir un groupement qui soit adéquat indépendamment du système. Par la suite, on utilise ses données pour avoir le partitionnement flou en utilisant la méthode FANNY, définie par Kaufman et Rousseeuw [40], et implémentée dans le logiciel S-Plus [31]. FANNY définit un ensemble de groupes et assigne chaque élément à chacun des groupes avec un certain degré d'appartenance.

Cette classification doit respecter deux contraintes: les degrés d'appartenance sont positifs et la somme des degrés d'appartenance d'un élément i à tous les groupements est égale à 1.

Si nous représentons les degrés d'appartenance obtenus en fonction du nombre d'invocations entre les classes, nous aurons comme illustré dans la Figure 17 trois ensembles flous. Ces ensembles sont les états des nœuds du réseaux bayésien qui représentent les invocations entre les classes du système.

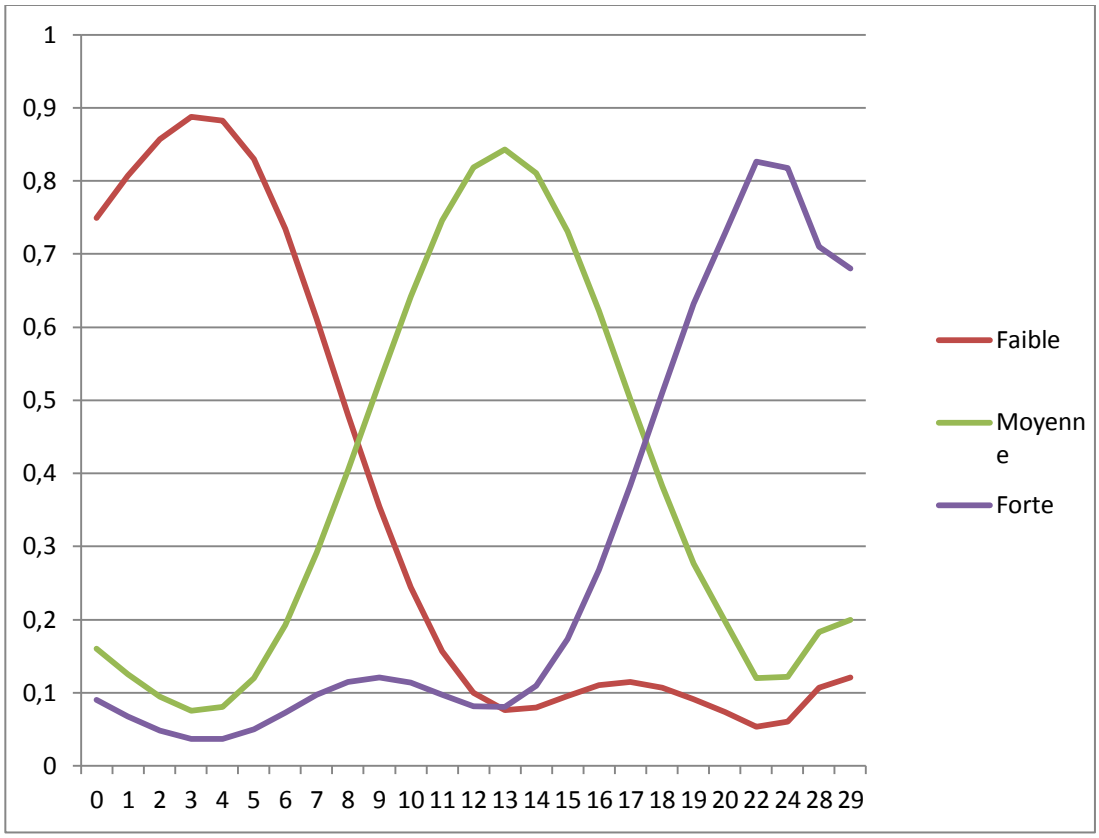

Figure 17: partitionnement flou pour le nombre d'invocations

Par la suite, afin de délimiter ces ensembles flous, on utilise la méthode proposée par Sahraoui et al. [61]. Il s'agit d'une méthode d'approximation utilisée dans le but de transformer les courbes en formes standard comme les trapèzes ou les triangles. Cette approximation est obtenue à travers l'intersection des tangentes aux courbes. Par conséquent, nous obtenons un graphe qui représente les grappes floues pour le nombre d'invocations comme illustrées dans la Figure 18.

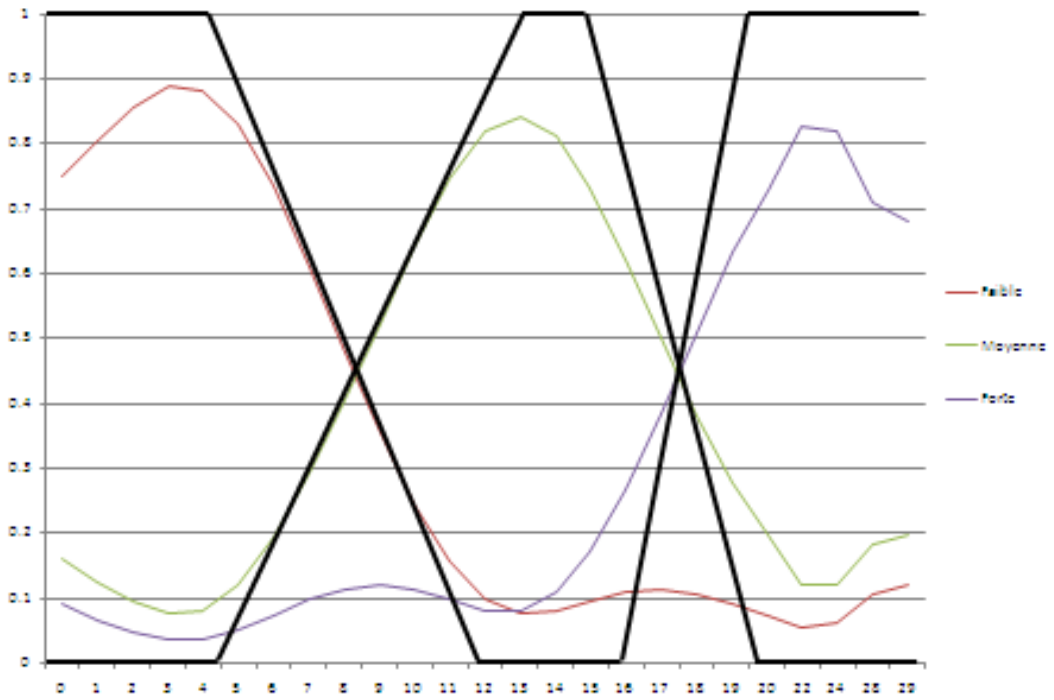

Figure 18: ensembles flous du nombre d'invocations

Une fois que le partitionnement flou et l'approximation des grappes sont réalisés, notre système est en mesure de déterminer pour chaque valeur d'invocation son degré d'appartenance aux types : faible, moyen ou fort. Ses degrés d'appartenances sont utilisés par la suite pour remplir la table de probabilité du nœud invocation du réseau bayésien lors de l'inférence.

4.3 Le protocole de validation

4.3.1 Collecte des données

Afin de tester l'efficacité de notre approche, nous appliquons le processus illustré dans la Figure 16 sur différents systèmes. En effet, dans ce chapitre nous vérifions si les réseaux que nous définissons et l'apprentissage que nous y appliquons nous permettent de prédire les changements. La mise en œuvre de notre approche se fait avec le logiciel WEKA [32]. Ce dernier nous permet de construire les structures des réseaux, d'effectuer l'apprentissage et d'appliquer l'inférence. En plus, son interface de programmation (API)

nous permet de l'intégrer à notre outil. Par conséquent, le système réalisé peut à la fois extraire les dépendances à partir du code et appliquer la l'inférence bayésienne pour analyser l'impact de changement. Afin de valider notre approche, nous avons choisi ces systèmes écrits en langage Java :

- Xerces [37]: une bibliothèque pour l'analyse syntaxique, la validation et la manipulation des documents XML (Extensible Markup Language).
- JFreeChart [33] : une bibliothèque qui aide les développeurs à réaliser des diagrammes selon leurs demandes.
- OpenJMail [30]: une interface de programmation (API) qui implémente des protocoles pour la messagerie et le courrier électronique.
- JFlex [29]: un générateur d'analyseurs lexicaux.
- EIRC [28]: un client Web pour la messagerie instantanée.

À travers ce choix de systèmes, nous avons essayé d'avoir un échantillon qui soit représentatif. Pour y parvenir, nous avons choisi des systèmes destinés à des domaines différents. En plus, ils ne sont pas du même type (bibliothèque, interface, client web). Enfin, les systèmes qu'on utilise pour collecter nos données sont de tailles variables. Le Tableau 3 résume les transitions de versions qu'on utilise pour chaque système ainsi que la taille de ces systèmes.

Xerces	**JFreeChart**	**OpenJMail**	**JFlex**	**EIRC**
1.0.1 → 1.4.4 (10 versions)	0.5.6 → 0.8.0 (8 versions)	1.0.1 → 1.0.9 (9 versions)	1.3.0 → 1.4.1 (8 versions)	1.0.1 → 2.0.0 (5 versions)
578 classes (version 1.4.4)	331 classes (version 0.8.0)	71 classes (versions 1.0.9)	59 classes (version 1.4.1)	69 classes (versions 2.0.0)

Tableau 3: les systèmes utilisés pour l'évaluation

Les cinq systèmes présentés vont servir à l'évaluation de l'approche probabiliste proposée. En effet, les données utilisées pour l'apprentissage sont obtenues en comparant chaque paire de versions successives de ces systèmes en utilisant Chianti. Nous obtenons alors les différents changements appliqués lors du passage de la version i à la version i+1, ainsi que les dépendances entre ces changements. Par la suite, nous identifions la classe où les changements dépendants doivent être appliqués. Enfin, nous récupérons les

dépendances entre la classe changée et celles où les changements dépendants sont appliqués pour obtenir les liens qui sont responsables de la propagation du changement.

4.3.2 Mesure de la performance dans un contexte probabiliste

L'évaluation de notre approche est basée sur deux aspects : la précision et le rappel. Dans une démarche probabiliste, ces deux termes se rapportent aux prédictions générées. Vu que le résultat de notre approche est une prédiction qui indique la probabilité qu'une classe soit affectée, nous utilisons un seuil pour simuler la décision d'un utilisateur. Ainsi, si le seuil est fixé à 0.5 alors toutes les classes dont la probabilité est supérieure à ce seuil sont prédites comme affectées avec une probabilité. Une fois qu'on attribue une probabilité à toutes les classes du système on peut calculer un taux de précision et de rappel comme suit :

$$Précision = \frac{\mathrm{VP}}{VP + FP} \qquad\qquad Rappel = \frac{\mathrm{VP}}{\mathrm{VP} + \mathrm{FN}}$$

VP : Le nombre de classes prédites affectées et qui le sont réellement (vrai positif).
FP : Le nombre de classes prédites affectées et qui ne le sont pas réellement (faux positif).
FN : Le nombre de classes prédites comme non affectées alors qu'elles le sont réellement (faux négatif).

Intuitivement la précision calcule le taux des bonnes prédictions et le rappel nous renseigne sur la complétude des prédictions obtenues. En effet, la précision représente, comme le montre la formule, la proportion des bonnes prédictions par rapports à l'ensemble des classes que notre approche prédit comme affectées. Quant à la formule du rappel, elle nous renseigne sur le pourcentage des classes que notre approche définit comme affectées par rapport à l'ensemble des classes qui le sont réellement. Afin d'évaluer notre approche, nous avons également utilisé une mesure d'efficacité globale, la F-measure de Van Risbergen [67]. L'intérêt de l'utilisation de la F-mesure est que celle-ci combine la précision et le rappel en une mesure unique. Ainsi elle peut nous renseigner sur l'efficacité de notre proche en termes d'exactitude et de complétude. Elle est définie comme suit :

$$\boldsymbol{F - mesure = \frac{2 * Précision * Rappel}{Précision + Rappel}}$$

4.3.3 Évaluation de l'approche proposée

Dans notre projet, nous proposons un ensemble de réseaux bayésiens qui prédisent la propagation des changements à travers les classes du système suite à un changement apporté à une classe. La qualité de nos prédictions exprimées en termes de précision et de rappel dépend de la structure des réseaux bayésiens utilisés et de l'apprentissage qui a été réalisé. À travers ce mémoire, nous essayons de répondre à deux questions :

- Question 1 : peut-on prédire l'impact de changements d'un système à partir des données sur ses versions antérieures **(Prédictions intra-système)**?
- Question 2 : peut-on prédire l'impact de changement d'un système avec des données provenant d'autres systèmes **(Prédictions inter-systèmes)**?

La première question peut se poser dans un environnement qui offre aux programmeurs l'historique du système à maintenir. Ainsi, les données d'apprentissage sont extraites à partir des versions précédentes pour analyser comment le changement pourrait se propager dans la version courante. Cependant, l'accès à ces données peut ne pas être possible si le système vient d'être mis en place ou pour des raisons de sécurité. Dans ce cas, nous explorons la deuxième question qui consiste à utiliser les données d'autres systèmes.

L'évaluation de notre proche se fait sur deux volets. Le premier consiste à comparer nos prédictions par rapport avec des seuils de confiance. Les classes prédites avec une probabilité supérieure à ces seuils sont considérées comme affectées. Le deuxième volet consiste à ordonner les classes selon la probabilité de d'impact. Ensuite, nous essayons de déterminer si les classes qui sont au début de la liste sont des bonnes prédictions. Cette façon permettrait au mainteneur de vérifier les classes qui sont le plus susceptibles de changer en premier lieu. Le parcours de la liste des classes affectées dépend du temps alloué à la maintenance. Dans notre validation, nous nous arrêtons au niveau des classes prédites avec une probabilité de 0.5. Ce seuil est utilisé à titre indicatif.

4.4 Prédictions intra-système

Afin de répondre à la première question, nous exécutons nos réseaux bayésiens sur différents changements. En effet, le processus commence par collecter les données relatives aux types des changements dans les (n-1) versions d'un système. Ces données sont utilisées pour réaliser l'apprentissage des réseaux. Par la suite, nous comparons les prédictions obtenues avec les données des changements réels que nous possédons pour la dernière version (version n). Ce processus est répété pour les différents systèmes utilisés. Le résultat de ce processus est l'attribution d'une probabilité d'impact pour toutes les classes du système. Les résultats de prédictions sont par la suite comparés à un seuil qui indique qu'une classe est prédite comme affectée ou pas. Nous avons varié le seuil (60%, 70% et 80%). Ainsi si une classe est prédite au dessous du seuil choisi alors est elle est considérée comme non affectée sinon elle l'est. Les résultats selon les trois seuils sont montrés dans le Tableau 4.

Les données collectées nous ont permis de tester des changements qui se rapportent à différentes granularités (classe, méthode et attribut). La liste de changements que nous prédisons avec notre approche est:

- CC: modification de la signature d'une classe (les classes dont elle hérite ou les interfaces qu'elle implémente).
- DC: suppression d'une classe
- AM: ajout d'une méthode
- CM: modification du corps d'une méthode ou sa signature.
- DM: suppression d'une méthode
- AF: ajout d'un attribut
- DF: suppression d'un attribut

	Système	Précision (60%)	Rappel (60%)	F - Mesure	Précision (70%)	Rappel (70%)	F - Mesure	Précision (80%)	Rappel (80%)	F - Mesure
CC	JFREECHART	1	1	1.0000	1	1	1.0000	1	1	1.0000
CC	XERCES	1	1	1.0000	1	1	1.0000	1	0.6667	0.8000
DC	JFREECHART	1	0.8235	0.9032	1	0.8235	0.9032	1	0.7059	0.8276
AM	EIRC	1	1	1.0000	1	0.5	0.6667	1	0.5	0.6667
AM	JFLEX	1	0.76	0.8636	1	0.64	0.7805	1	0.4	0.5714
AM	JFREECHART	1	1	1.0000	1	1	1.0000	-	-	-
AM	OPENJMAIL	0.9375	1	0.9677	0.9375	1	0.9677	0.9375	1	0.9677
AM	XERCES	1	1	1.0000	1	0.6667	0.8000	-	-	-
CM	EIRC	1	0.9473	0.9729	1	0.9474	0.9730	-	-	-
CM	JFREECHART	1	1	1.0000	1	0.5	0.6667	-	-	-
CM	XERCES	1	1	1.0000	-	-	-	-	-	-
CM	OPENJMAIL	1	1	1.0000	1	1	1.0000	1	1	1
CM	JFLEX	0.9696	0.5818	0.7272	0.9474	0.3273	0.4865	0.9474	0.3273	0.4865
DM	EIRC	1	0.5556	0.7143	1	0.3333	0.5000	1	0.2222	0.3636
DM	JFREECHART	0.9473	0.6923	0.8000	0.9474	0.6923	0.8000	0.9444	0.6538	0.7727
DM	XERCES	0.9166	0.8462	0.8800	0.9167	0.8462	0.8800	1	0.1538	0.2666
AF	JFLEX	0.8461	0.9167	0.8800	0.8462	0.9167	0.8800	0.8333	0.8333	0.8333
DF	JFREECHART	0.94	0.79	0.85	0.94	0.79	0.85	0.94	0.79	0.85
DF	XERCES	1	1	1.0000	1	1	1.0000	-	-	-

Tableau 4: précisons et rappels intra-système

Le constat principal qu'on peut faire à partir de ces résultats est que les réseaux bayésiens proposés et l'apprentissage appliqué nous permettent de prédire les changements considérés avec une très bonne précision (>0,83) pour les cinq systèmes considérés. On peut donc affirmer qu'à des rares exceptions, les prédictions correspondent à la réalité même avec un seuil de 60%. Cependant, le rappel varie considérablement. Le rappel exprime la portion des classes trouvées parmi celles qui devraient être détectées. Le rappel dans nos expériences est généralement acceptable (>0,6) dans la majorité des cas de figure. Cependant, si on lève le seuil à 80%, le rappel diminue, car il existe des classes auxquelles notre système attribue une probabilité d'impact inférieure à 80%. Ceci ne remet pas pour

autant en cause les structures des réseaux bayésiens, car pour le même type de changement on peut avoir des rappels plus élevés avec d'autres systèmes. Par exemple, notre approche a un rappel de 33% si on veut appliquer une suppression de méthode (DM) sur le système EIRC. Le rappel atteint 69% pour la même configuration avec le système JFreeChart et 84% pour XERCES. Ainsi, ce taux de rappel s'explique par le fait que durant la dernière version d'EIRC, le changement ne s'est pas propagé comme dans les versions antérieures de ce système. Nous proposons pour ce type de changement un réseau bayésien qui comprend comme nœuds d'entrée: l'héritage, l'invocation et l'agrégation. Dans les versions utilisées pour l'apprentissage, ce changement se propageait à travers le lien de l'héritage ou à travers la combinaison d'une agrégation et d'une invocation forte. Cependant ce changement se propage dans la dernière version (celle que nous testons) à travers le lien d'agrégation et une invocation faible. En plus ce rappel s'explique aussi par le nombre de versions utilisées pour tester ce système qui n'est pas élevé (4 versions).

La colonne de la F-Mesure indique le compromis entre la précision et le rappel pour les différents types de changement. Cette mesure montre que notre approche affiche des résultats de plus de 75% dans la majorité des cas. Ce taux est tout à fait acceptable dans un contexte de priorisation de l'effort de maintenance ou des choix parmi des alternatives de conception.

Les réseaux bayésiens nous permettent d'ajouter ces changements dans les fichiers d'apprentissage afin de cerner ces cas particuliers qui causent la diminution du rappel, mais nous avons décidé d'exclure cette alternative et de faire recours uniquement aux données extraites à partir des anciennes versions.

Les cases du tableau qui comprennent le symbole «-» correspondent à deux cas: pour la précision, on a TT+FT= 0 et pour le rappel TT+TF = 0. Ces cases correspondent au cas où les prédictions du réseau bayésien sont inférieures au seuil utilisé (60%,70%,80%).

Malgré que nous utilisions plusieurs systèmes, il arrive souvent que nous ne trouvions pas assez de données pour faire les tests, car l'apprentissage requiert une quantité d'information qui permet de calibrer les tables de probabilités de nœuds. Cette rareté d'information s'explique par le fait qu'on exclut de l'ensemble des classes affectées celles qui le sont suite à un impact local. En effet, nous considérons qu'une classe doit avoir la priorité de vérification suite à son changement.

Dans la réalité les mainteneurs ont des ressources limitées en termes de temps. Par conséquent, ils essaient de limiter leurs interventions. Dans cette perspective de maintenance, on se pose la question sur la capacité de notre approche à aider les mainteneurs dans cette quête d'efficacité. Afin de vérifier cette possibilité, on ordonne les classes par leurs probabilités d'impact dans un ordre croissant. Par la suite, c'est aux mainteneurs de choisir à quel niveau ils vont arrêter de consulter les classes affectées en fonction de leurs moyens. La qualité de notre approche va dépendre de l'ordre d'apparition des classes dans la liste. Si les premiers éléments dans la liste sont réellement affectés, nous allons pouvoir aider les mainteneurs à cibler les classes à vérifier, par contre si nos réseaux prédisent une classe affectée avec une grande probabilité alors qu'elle ne l'est en réalité, notre approche risque de dissiper les efforts des mainteneurs. Si nous visualisons l'évolution de la précision et du rappel en fonction des classes inspectées, l'allure idéale sera celle présentée dans la Figure 19. En effet, un tel graphe signifie que la précision est constante (les classes au début de la liste sont réellement affectées). En plus, le rappel augmente continuellement jusqu'à atteindre 100% (toutes les classes affectées sont détectées).

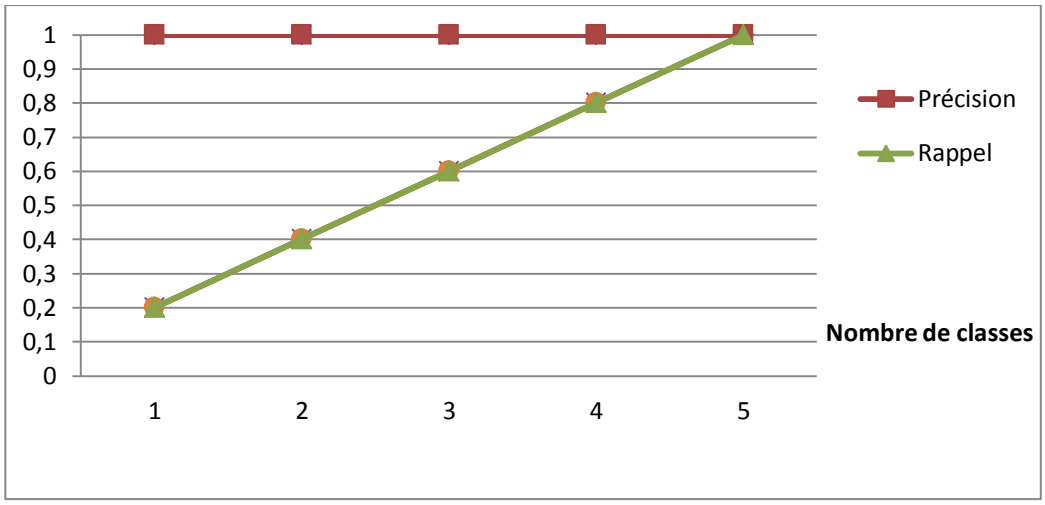

Figure 19: allure idéale pour la précision et le rappel

Afin d'exprimer ce volet de validation, nous avons choisi un seuil de 50%. Ainsi, les classes prédites au-delà de ce seuil sont considérées comme affectées. Par la suite, on compare nos prédictions aux données réelles et on calcule la précision et le rappel d'une façon incrémentale : on considère le premier élément, ensuite les deux premiers jusqu'à la dernière classe, dont les probabilités d'impact, dépassent les 50%.

La quantité de données collectées dépend des systèmes utilisés et des modifications appliquées à travers ses versions. Ceci explique que nous ne possédons pas de données suffisantes pour certaines combinaisons système- type de changement.

4.4.1 Modification d'une classe (CC)

La modification de la signature d'une classe est un changement que les mainteneurs n'appliquent pas souvent. Ceci explique le fait que nous n'avons pas trouvé beaucoup de cas de propagation dû à ce changement dans les différentes versions de systèmes que nous avons étudié. Les allures des graphiques illustrées dans la Figure 20 et la Figure 21 montrent que nous arrivons à bien prédire ce type de changement.

Bien que les deux graphiques se ressemblent, notre approche prédit les modifications de classes dans JFreeChart mieux que dans XERCES. En effet, les classes dans JFrechart sont prédites avec une probabilité entre 0.833 et 0.916. Dans Xerces, les classes sont prédites avec une probabilité entre 0.75 et 0.833.

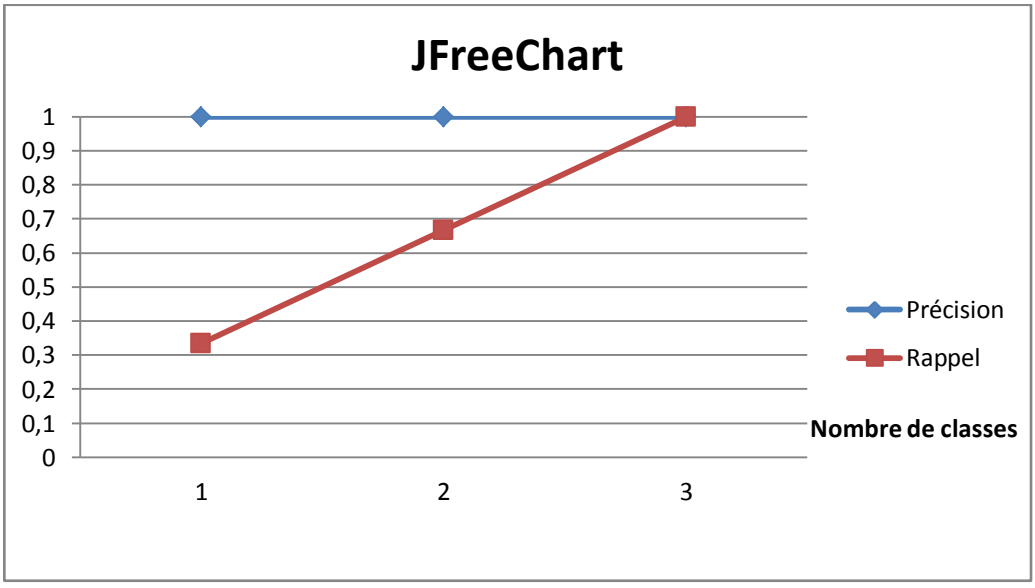

Figure 20: JFreeChart: précision et rappel pour le changement de classe (CC)

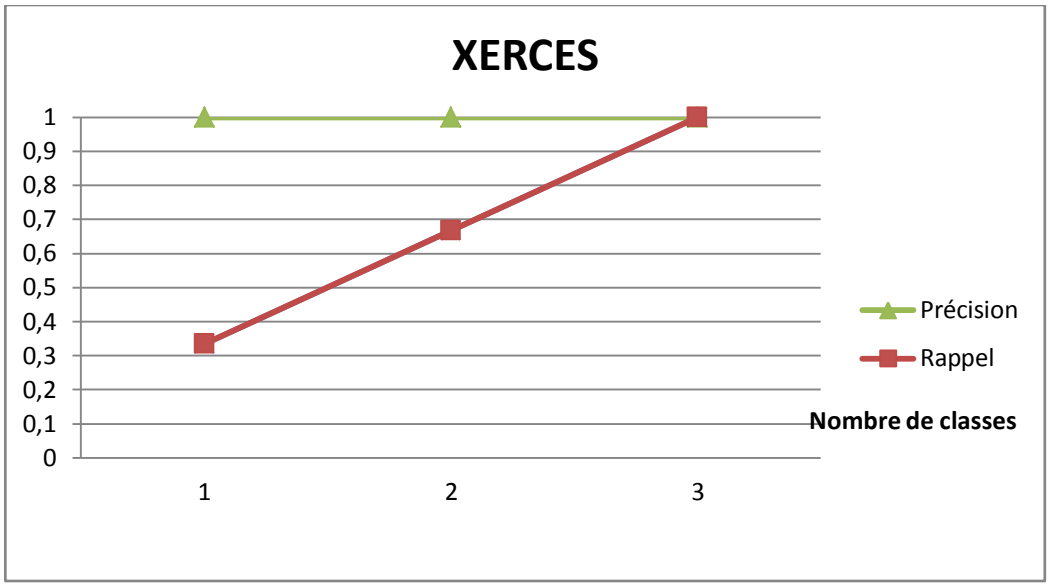

Figure 21: XERCES: précision et rappel pour le changement de classe (CC)

4.4.2 Suppression d'une classe (DC)

Les versions utilisées dans ce projet ne comprennent pas beaucoup de suppression de classe. En effet, la suppression d'une classe est un changement que les programmeurs n'effectuent pas souvent, car en supprimant une classe, les effets de ce type de changement peuvent être néfastes.

Notre système prédit bien ce changement pour le système JFreeChart. En effet, les l'allure de la courbe, présentée dans la Figure 22, se rapproche de l'idéal attendu. Les 14 premières classes de notre prédiction ont une probabilité > 50% et elles sont des classes qui sont réellement affectées. Ainsi, on peut affirmer que la combinaison (H: héritage, G: Agrégation, A: Association, I: Invocation) nous permet de bien prédire les classes affectées suite à une suppression de classe. On remarque aussi que le rappel dépasse les 80%. Notre système n'a pas pu prédire deux cas de propagation de changement. Le premier est l'impact sur la classe *com.jrefinery.chart.ChartFactory* suite à la suppression de la classe *com.jrefinery.chart.LinePlot*. Le second cas non détecté par notre réseau est l'impact sur la classe *com.jrefinery.chart.StackedVerticalBarRenderer* suite à la suppression de la classe *com.jrefinery.chart.BarPlot*. Ceci s'explique par le fait, que ces deux couples de classes ne soient pas reliées par des liens d'association, d'agrégation, d'héritage ou d'invocation forte comme pour les cas similaires de propagation rencontrés dans les versions antérieures du

système JFreeChart. En effet, la propagation s'est produite pour ces deux cas à cause d'un lien d'invocation faible.

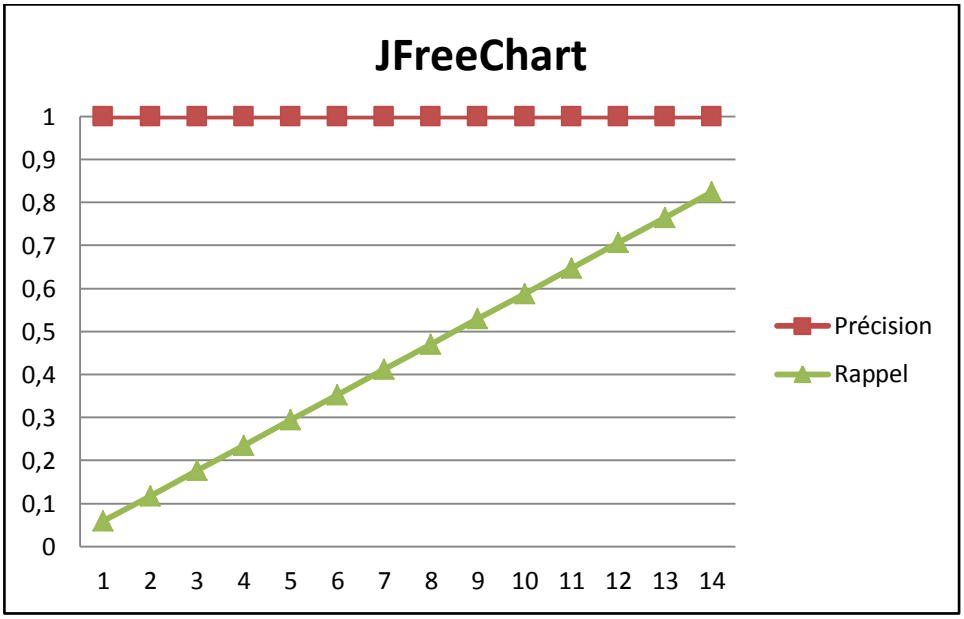

Figure 22: JFreeChart: précision et rappel pour le changement de classe (DC)

4.4.3 Ajout d'une méthode (AM)

Avec la structure de réseau (H,A,I), notre approche arrive à bien prédire les classes affectées suite à un ajout de méthode pour les systèmes JFlex et Xerces. Les allures de courbes de ces deux systèmes illustrés dans la Figure 23 et la Figure 24 se rapprochent de l'allure idéale. Pour JFlex, les classes qui sont classées au début de la liste des classes affectées sont des bonnes prédictions. Ceci se reflète par une précision constante égale à 100%. Les résultats montrent que le réseau détecte 80% des classes qui sont réellement affectées. Des classes comme JFlex.LexScan et JFlex.StateSet sont prédites comme non affectées alors qu'elles le sont en réalité. Ces prédictions sont obtenues, car ces classes sont reliées avec les classes changées uniquement par une invocation faible. Dans les fichiers d'apprentissage, ce cas de figure ne correspondait pas généralement à une propagation et ceci explique le fait que notre réseau attribue à ces cas une probabilité inférieure à 0.5.

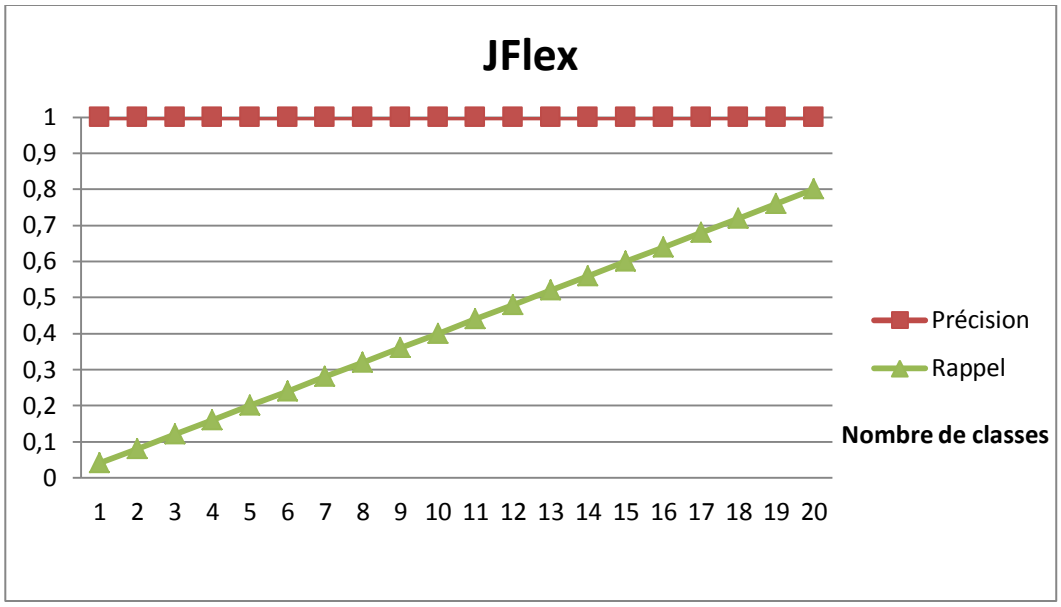

Figure 23: JFlex: précision et rappel pour l'ajout de méthode (AM)

Concernant le système XERCES, les trois premières classes prédites sont réellement affectées avec une probabilité supérieure à 0,625. Cependant, la dernière classe dans la liste ne l'est pas. En effet, selon nos prédictions, l'ajout d'une méthode dans la classe *org.apache.xerces.dom.AttrImpl* va impacter la classe *org.w3c.dom.DOMImplementation* avec une probabilité de 0.5 alors que ce n'est pas le cas en réalité. Ce cas d'erreur n'est pas grave une probabilité de 0.5 indique que le réseau n'est pas certain de sa prédiction. En plus, nous aurions pu éviter ce cas si nous limitons notre recherche de classe affectées si nous atteignons un rappel de 100% (toutes les classes affectées sont identifiées) comme fut le cas avec Xerces.

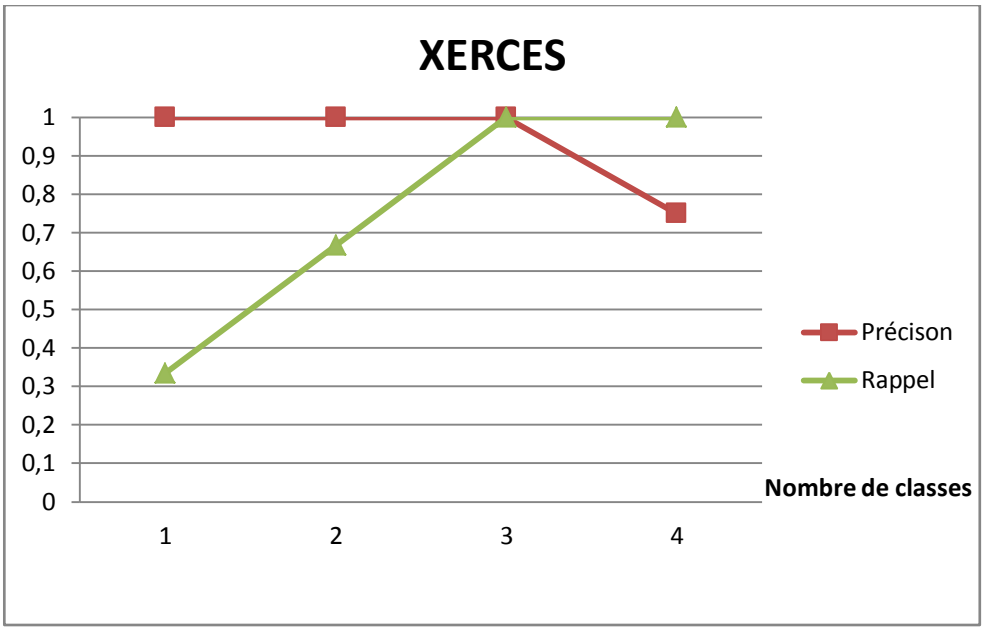

Figure 24: XERCES: précision et rappel pour l'ajout de méthode (AM)

Avec le système OpenJmail, le graphique illustré dans la Figure 25 commence différemment de l'allure attendue. Par la suite, l'allure commence à tendre vers la forme que nous espérons avoir. La classe *net.larsan.email.stream.B_OutputStream* est prédite comme affectée suite à un changement dans la classe *net.larsan.email.Body*. Ceci n'est pas le cas en réalité. Ce cas d'erreur est induit, car ces deux classes sont reliées par un lien d'héritage. Ce type de lien propage ce type de changement dans les versions antérieures utilisées lors de l'apprentissage. À part le premier cas, les autres prédictions de notre réseau sont correctes. Ceci se traduit par une augmentation graduelle de la précision et du rappel qui finit par atteindre 100%.

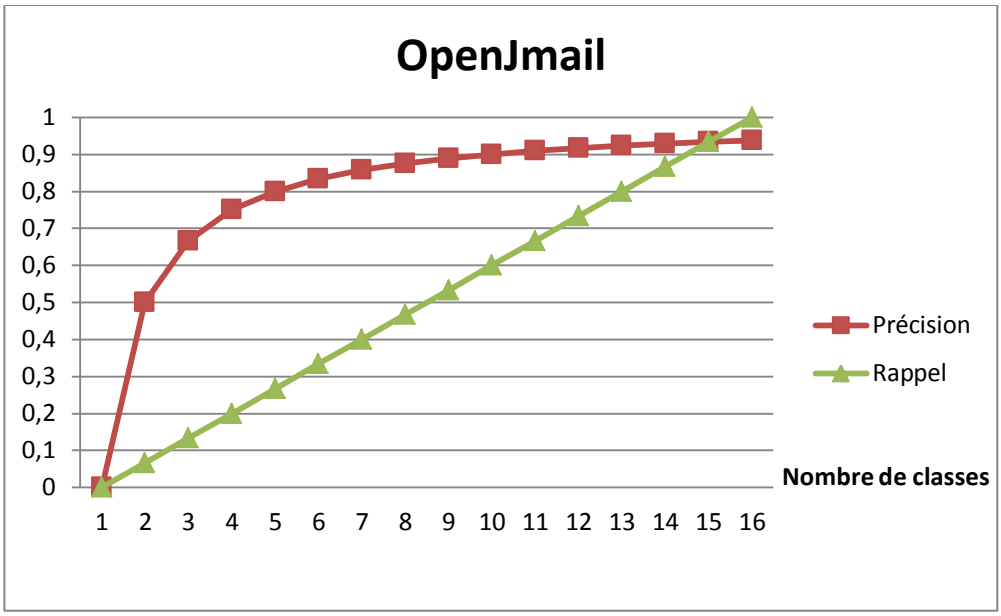

Figure 25: OpenJmail: précision et rappel pour l'ajout de méthode (AM)

4.4.4 Modification d'une méthode (CM)

Le changement de méthode est le changement que le mainteneur effectue le plus, car c'est une façon de modifier le fonctionnement du système pour qu'il réponde à des nouveaux besoins. Dans notre expérience, ce changement a été très bien prédit dans le système EIRC vu qu'il s'est propagé comme dans les versions précédentes (soit à travers le lien d'héritage soit avec la combinaison du lien d'association et d'agrégation). Ceci se reflète dans le graphique illustré dans la Figure 26. Ce graphique indique un taux de précision de 100% et un taux de rappel égal à 94%.

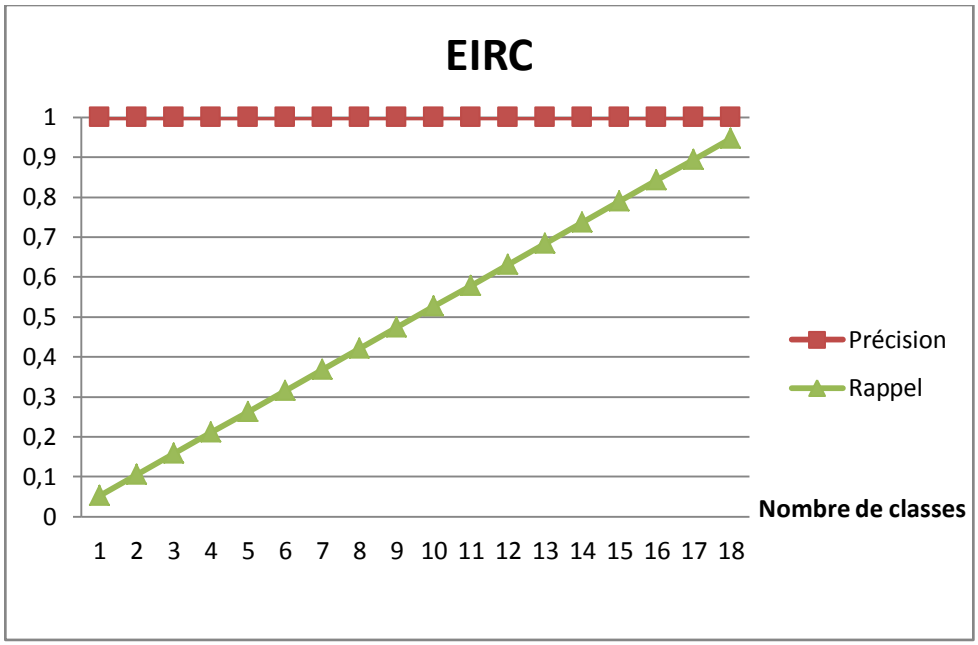

Figure 26: EIRC: précision et rappel pour le changement de méthode (CM)

Pour le système JFlex, l'allure du graphique obtenue dans la Figure 27 se rapproche de l'allure que nous espérons avoir. Les prédictions obtenues présentent trois cas ou notre réseau s'est trompé sur un ensemble de 39 prédictions. Suite à un changement de méthode dans la classe *JFlex.StdOutWriter,* on trouve parmi les classes prédites comme affectées la classe *JFlex.CharSetEnumerator*. Notre système a été induit en erreur, car à la combinaison (Pas d'héritage, Pas d'agrégation, Association, Invocation faible) dans les fichiers d'apprentissage correspondait généralement une classe affectée dans les versions précédentes. Ceci indique que le lien d'association entre les classes propageait ce type de changement dans les versions (n-1), mais ce n'est pas le cas pour la version de test (version n). Par ailleurs, notre système s'est trompé aussi en prédisant qu'un changement dans la classe *JFlex.CharClasses* va affecter les classes *JFlex.Emitter* et *java_cup.runtime.Scanner* avec une probabilité de 0.5678. Cette deuxième erreur, qui a causé la deuxième chute de précision dans le graphique, est due au fait que notre système a appris à travers les versions antécédentes qu'une forte invocation propage généralement le changement et ce ne fut pas le cas pour ce couple de classes. Nous pensons que c'est dû au fait que les appels de méthodes entre ces couples de classes ne touchent pas à la méthode modifiée.

Malgré les cas d'erreurs détectés avec le système JFlex, il faut noter qu'il s'agit de trois sur quarante classes prédites comme affectées. Ainsi, le taux de précision reste au-

delà de 90%. Par ailleurs, même si le taux de rappel atteint 70%, les autres classes affectées sont classées juste en dessous du seuil de 50%.

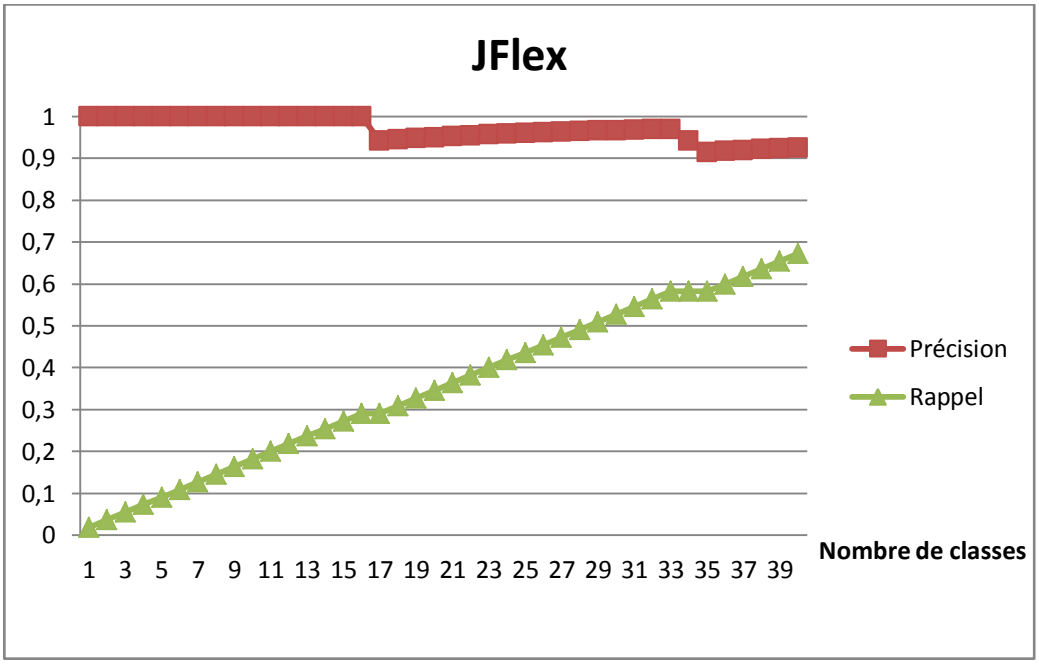

Figure 27: JFlex: précision et rappel pour le changement de méthode (CM)

4.4.5 Suppression d'une méthode (DM)

Pour le système EIRC, les sept premières classes sont correctement prédites. Ceci explique la précision constante et le rappel croissant dans la première partie du graphique illustré dans la Figure 28. Par contre, à partir de la 8ème classe, les classes sont prédites comme affectées alors qu'elles ne le sont pas réalité. En effet, notre réseau leur attribue une probabilité de 0.5. Ces classes sont reliées par une invocation forte. Ainsi, notre réseau reste indécis quant à cette propagation et décide de leur attribuer une probabilité qui va laisser les mainteneurs se demander s'il vaut la peine d'aller consulter ces classes ou pas. Nous constatons que le rappel atteint 0.77. En consultant les résultats de prédiction, nous avons trouvé les deux cas qui manquaient pour que nous arrivions à 100% de rappel. Le premier consiste à l'impact qu'a subi la classe *EIRC* suite à une suppression de méthode dans la classe *Commands*. Notre réseau attribue une probabilité de 0.49. Le deuxième cas de mauvaise prédiction est celui de l'impact identifié dans la classe *EIRC* induit par un nombre d'invocations faible avec la classe *netscape.security.PrivilegeManager*.

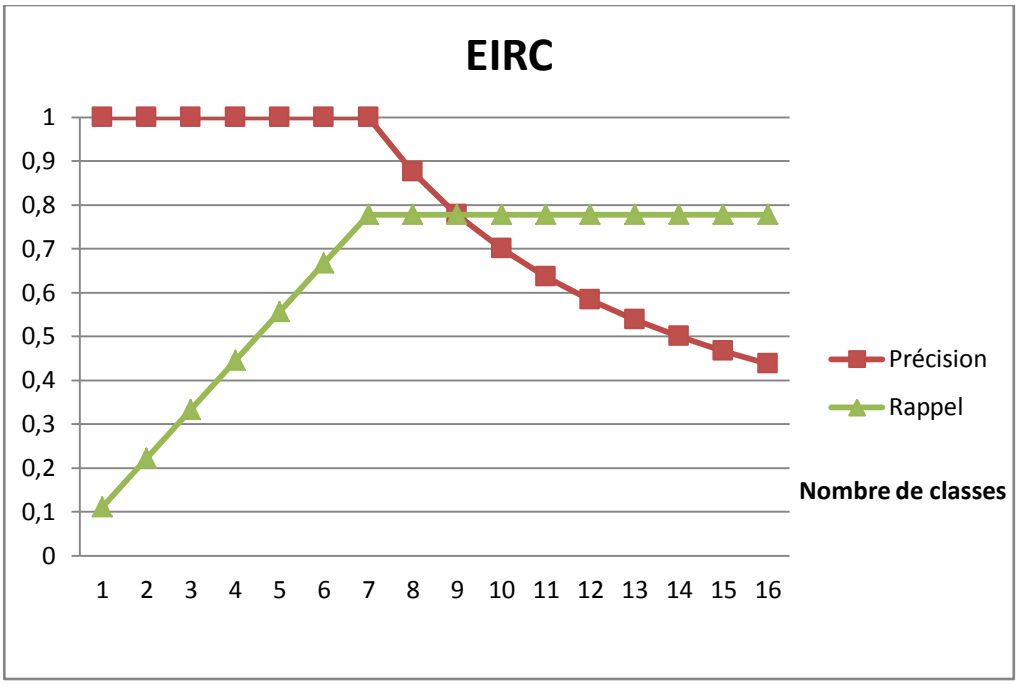

Figure 28: EIRC: précision et rappel pour la suppression de méthode (DM)

Pour JFreeChart, la prédiction est meilleure que celle dans EIRC. En effet, la Figure 29 montre que sur les 19 classes qui sont prédites comme affectées, 18 le sont réellement. La seule prédiction erronée, qui engendre la diminution de précision comme le montre la Figure 29, est quand le réseau prédit que la suppression de méthode dans la classe *com.jrefinery.chart.BarPlot* pourrait affecter la classe *com.jrefinery.chart.LegendItem*. En effet, le lien d'héritage propage généralement ce type de changement, mais ce ne fut pas le cas pour le couple de classe mentionné. Les classes affectées et qui ne sont pas détectées par notre réseau sont reliées par des invocations faibles. Ceci n'était pas le cas dans les versions antérieures où la propagation se produisait généralement à travers les liens d'héritage et d'agrégation.

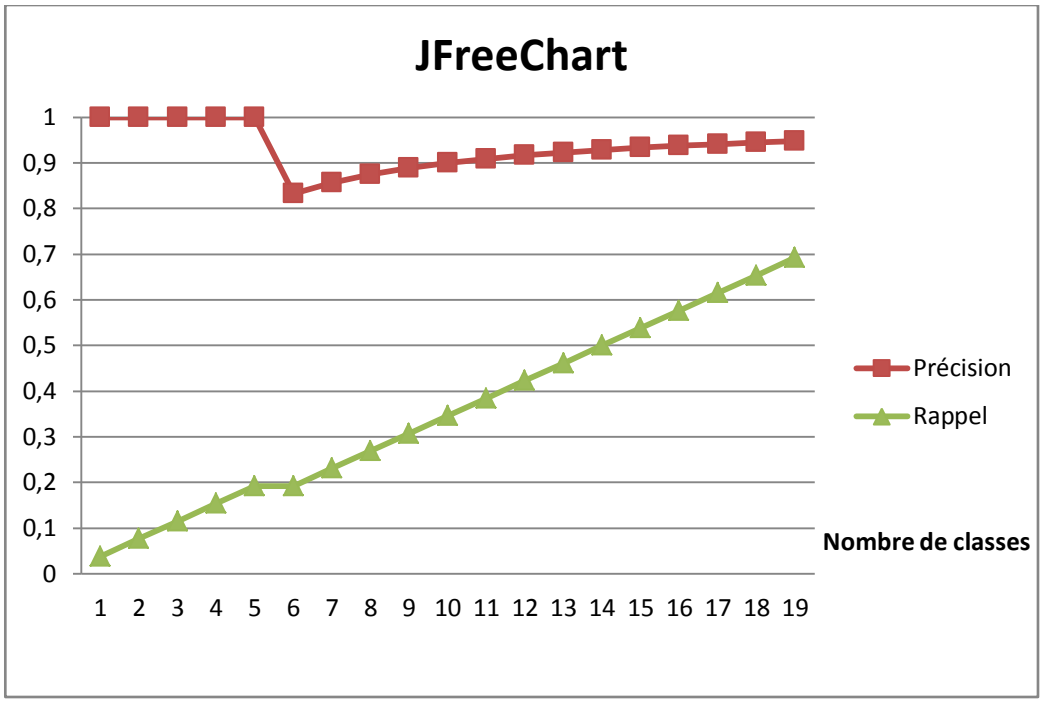

Figure 29: JFreeChart: précision et rappel pour la suppression de méthode (DM)

4.4.6 Ajout d'un attribut (AF)

L'ajout d'un attribut propage rarement un changement dans les classes du système. Nous n'avons trouvé ce type de cas que dans le système JFlex. En effet, diverses classes sont affectées lors du passage de la version 1.3.5 à la version 1.4.1. Ces classes sont généralement liées à la classe changée par un lien d'association et par une invocation forte. La Figure 30 montre que sur les 13 prédictions de notre réseau, deux sont erronées. Ces deux cas sont causés par un des liens que nous pensons responsables de la propagation de ce type de changement. En effet, le premier cas correspond au couple de classe (*JFlex.gui.GeneratorThread*, *JFlex.gui.MainFrame$4*) qui est relié par un lien d'association. Le second cas de figure est celui du couple (*JFlex.gui.MainFrame*, *JFlex.NFA*) qui est relié par une association forte.

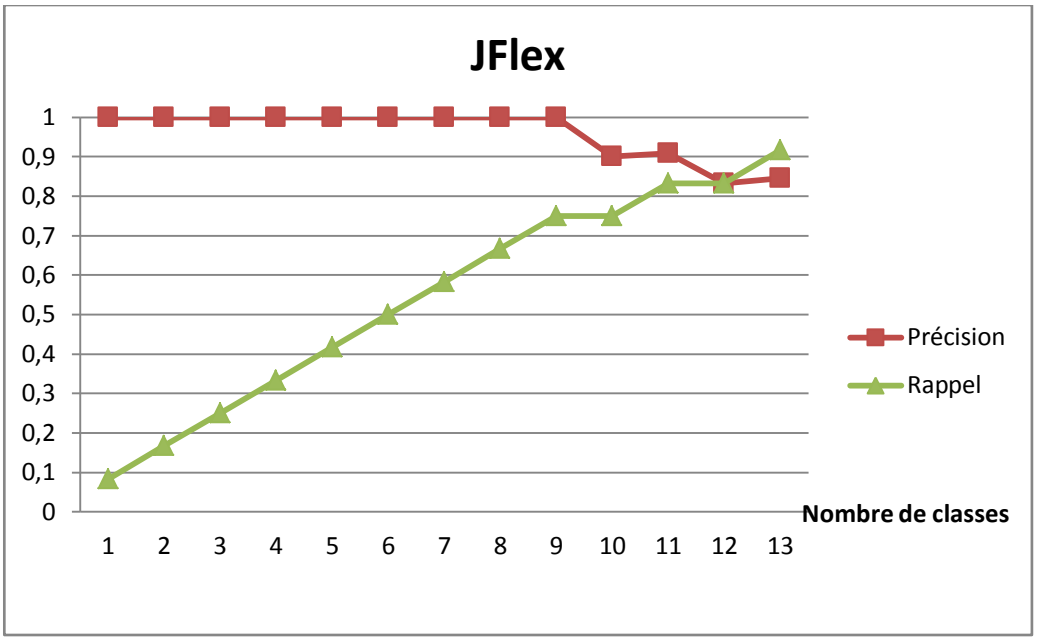

Figure 30: JFlex: précision et rappel pour l'ajout d'un attribut (AF)

4.4.7 Suppression d'un attribut (DF)

Les allures de la Figure 31 et de la Figure 32 montrent que la suppression d'un attribut est un changement que notre approche a très bien cerné avec une précision constante et un rappel croissant pour les systèmes EIRC et XERCES.

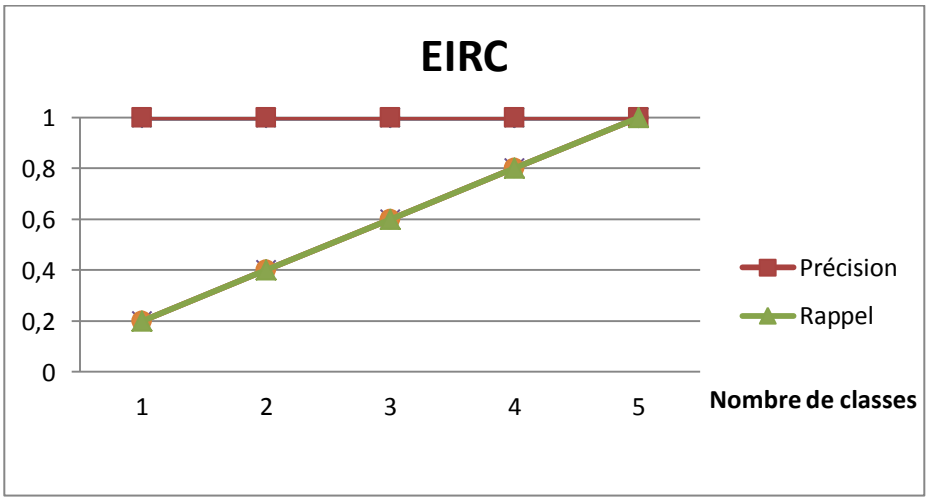

Figure 31: EIRC: précision et rappel pour la suppression d'un attribut (DF)

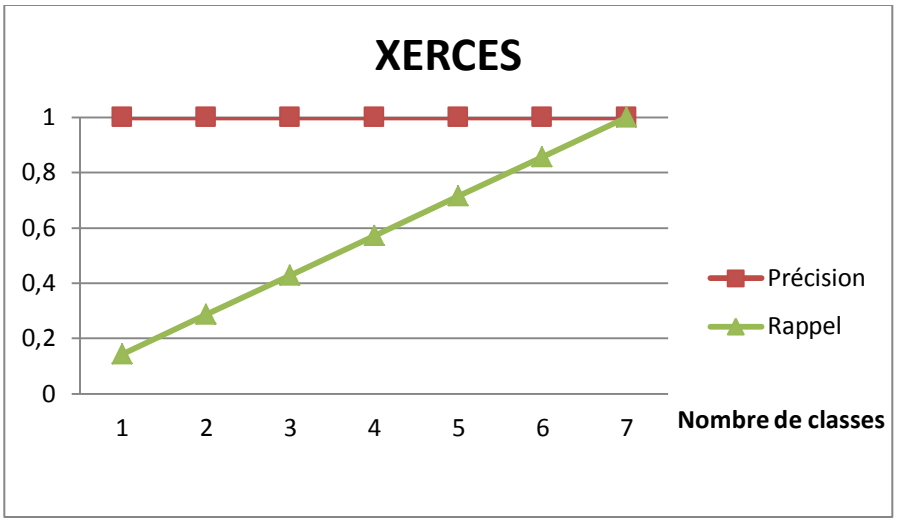

Figure 32: XERCES: précision et rappel pour la suppression d'un attribut (DF)

Cependant, pour le système JFreeChart la précision illustrée dans la Figure 33 affiche un petit déclin vers la fin du graphique. En effet, ces dernières classes induisent notre système en erreur vu qu'elles sont reliées aux classes modifiées par des liens d'héritage. Mais, le réseau leur attribue une probabilité inférieure aux autres, car le nombre de liens est inférieur pour ces classes par rapport aux classes qui sont réellement affectées. Nous remarquons aussi qu'il manque quelques classes à nos prédictions pour atteindre 100% de rappel. En effet, des classes comme *com.jrefinery.chart.demo.JFreeChartDemo* devraient être identifiées comme affectées suite à la suppression d'un des attributs de la classe *com.jrefinery.chart.JFreeChartConstants*. Notre réseau n'attribue pas une grande probabilité à ces classes, car la propagation s'est produite uniquement à cause des invocations reliant ces classes.

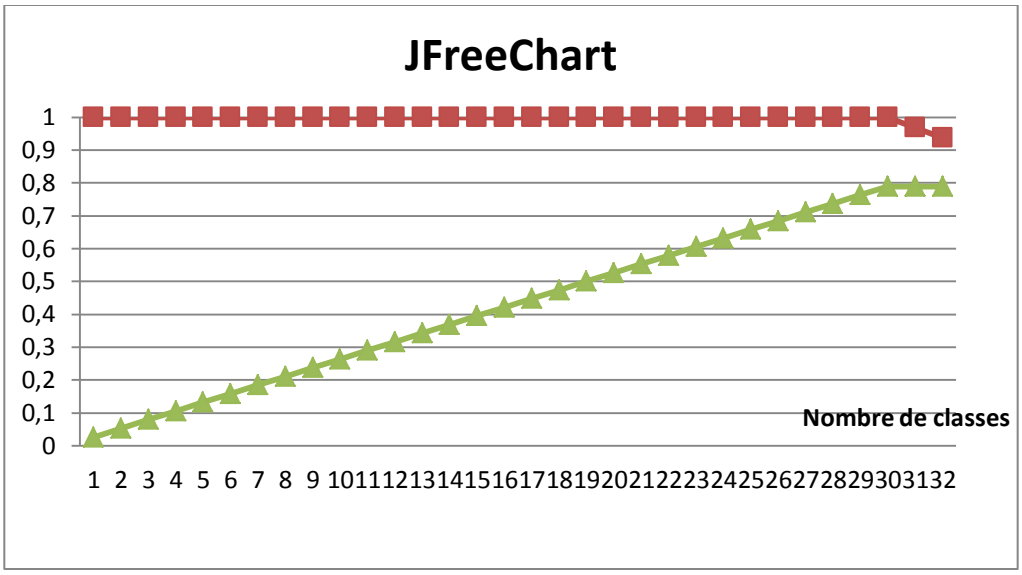

Figure 33: JFreeChart: précision et rappel pour la suppression d'un attribut (DF)

4.5 Prédictions inter-systèmes

Dans la deuxième partie de notre évaluation, on teste la capacité de notre approche à prédire des changements en utilisant des données d'apprentissage extraites d'autres systèmes. Le Tableau 5 résume la précision, le rappel et la F-Mesure pour différents seuils (60%, 70%, 80%). Les résultats obtenus sont encourageants vu que les données utilisées dans l'apprentissage sont complètement externes. En effet le taux de précision est généralement au-delà de 77% (sauf pour le système JFeeChart, car il possède plusieurs versions qui peuvent aider à prédire les autres systèmes). Le taux de rappel est inférieur en moyenne par rapport au premier scénario. En effet, il se trouve que certains systèmes ont des spécificités de propagations que ne possèdent pas les systèmes utilisés dans l'apprentissage. En d'autres termes les classes qu'on prédit comme affectées souvent le sont, mais il se peut qu'on n'arrive pas à identifier toutes les classes affectées. Les valeurs de la F-Mesure, bien qu'elles soient affectées par les valeurs du rappel, restent encourageantes pour plusieurs combinaisons de systèmes et de types de changements.

	Système	*Précision (60%)*	*Rappel (60%)*	*F - Mesure*	*Précision (70%)*	*Rappel (70%)*	*F - Mesure*	*Précision (80%)*	*Rappel (80%)*	*F - Mesure*
DC	EIRC	0.8571	0.5714	0.6857	0.8571	0.5714	0.6857	0.8182	0.4286	0.5625
DC	JFREECHART	0.9565	0.3729	0.5366	1	0.2203	0.3611	1	0.0847	0.1562
CM	EIRC	1	0.8621	0.9259	1	0.8621	0.9259	1	0.7586	0.8627
CM	JFREECHART	1	0.0294	0.0571	1	0.0294	0.0571	1	0.0294	0.0571
CM	XERCES	1	0.2222	0.3636	1	0.2222	0.3636	1	0.2222	0.3636
CM	OPENJMAIL	1	1	1.0000	1	1	1.0000	1	1	1.0000
CM	JFLEX	1	0.2199	0.3605	1	0.2199	0.3605	1	0.2199	0.3605
DF	EIRC	0.8	0.4444	0.5714	0.8	0.4444	0.5714	0.8	0.4444	0.5714
DF	JFREECHART	0.6233	0.5688	0.5948	0.5846	0.475	0.5241	0.55	0.4125	0.4714
CC	EIRC	1	1	1.0000	1	1	1.0000	1	1	1.0000
CC	JFREECHART	1	0.3333	0.5000	1	0.2464	0.3954	1	0.058	0.1096
CC	XERCES	1	0.4	0.5714	1	0.4	0.5714	1	0.4	0.5714
AM	EIRC	0.9167	1	0.9565	0.9167	1	0.9565	0.9167	1	0.9565
AM	JFREECHART	0.7778	1	0.8750	0.7778	1	0.8750	0.7778	1	0.8750
AM	XERCES	0.8	0.8	0.8000	0.8	0.8	0.8000	0.8	0.8	0.8000
AM	OPENJMAIL	0.95	1	0.9744	0.95	1	0.9744	-	-	-
AM	JFLEX	1	0.4217	0.5932	1	0.4217	0.5932	1	0.4217	0.5932
DM	EIRC	1	0.96	0.9796	1	0.96	0.9796	1	0.96	0.9796
DM	JFREECHART	0.9792	0.7287	0.8356	0.9792	0.7287	0.8356	0.9792	0.7287	0.8356
DM	XERCES	0.9412	0.6667	0.7805	0.9412	0.6667	0.7805	0.9412	0.6667	0.7805

Tableau 5: précisions et rappel inter-systèmes

Comme pour les tests intra-système, nous nous sommes intéressés à la capacité de notre approche à aider les mainteneurs pour accomplir leurs tâches. Nous classons nos

prédictions dans un ordre décroissant et nous calculons l'évolution de la précision et du rappel au fur et à mesure que le mainteneur choisit de vérifier une classe. Nos fixons un seuil de 0.5 comme point d'arrêt : les classes prédites au-delà de ce seuil sont considérées comme affectées. Ce choix de seuil peut varier selon les ressources disponibles aux mainteneurs. Si la classe prédite comme affectée l'est réellement, la précision reste constante ou augmente et le rappel croît. Dans le cas contraire, la précision diminue et le taux de rappel reste inchangé, car on n'a pas détecté une nouvelle classe de l'ensemble affecté.

4.5.1 Modification d'une classe (CC)

Les courbes illustrées dans la Figure 34 et la Figure 35 montrent que la modification de classe est un changement généralement bien capté par notre système. Ceci se traduit par une précision constante. En plus, nous pensons que nos prédictions peuvent être meilleurs dans le cas de JFreeChart car parmi les cas détectés il existe 29 dont la probabilité est de 0,5, car ils correspondent à une combinaison (H, Strong). Cette combinaison n'existe que très peu dans les fichiers des autres systèmes et ceci explique le fait que notre système ne lui attribue pas une grande probabilité alors que nous savons qu'il y aurait très probablement un impact. Le rappel atteint 0.666 dans le cas de XERCES. En effet, notre réseau n'a pas détecté les effets des changements apportés à des classes comme *org.apache.xerces.dom.EntityImpl* ou *org.apache.xerces.dom.ElementImpl* sur des classes comme *org.apache.xerces.parsers.DOMParser* et *org.apache.xerces.dom.NodeImpl*. Ces couples de classes sont reliés pas des liens d'invocations fortes. Ce type de propagation ne figure pas dans les autres systèmes utilisés pour l'apprentissage. Avec le système JFreeChart, le rappel est de l'ordre de 0.826. Il manque à nos prédictions des classes comme *com.jrefinery.chart.XYPlot* qui a été prédite avec une probabilité de 0.499 suite au changement apporté à la classe *com.jrefinery.chart.StandardXYItemRenderer*.

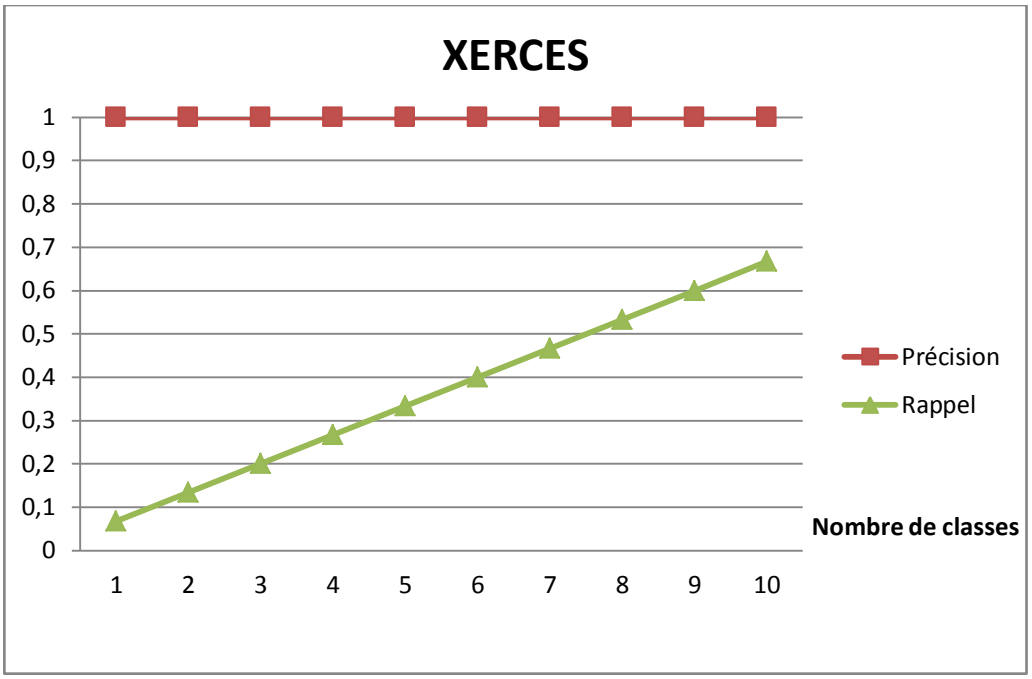

Figure 34: XERCES: précision et rappel pour le changement d'une classe (CC)

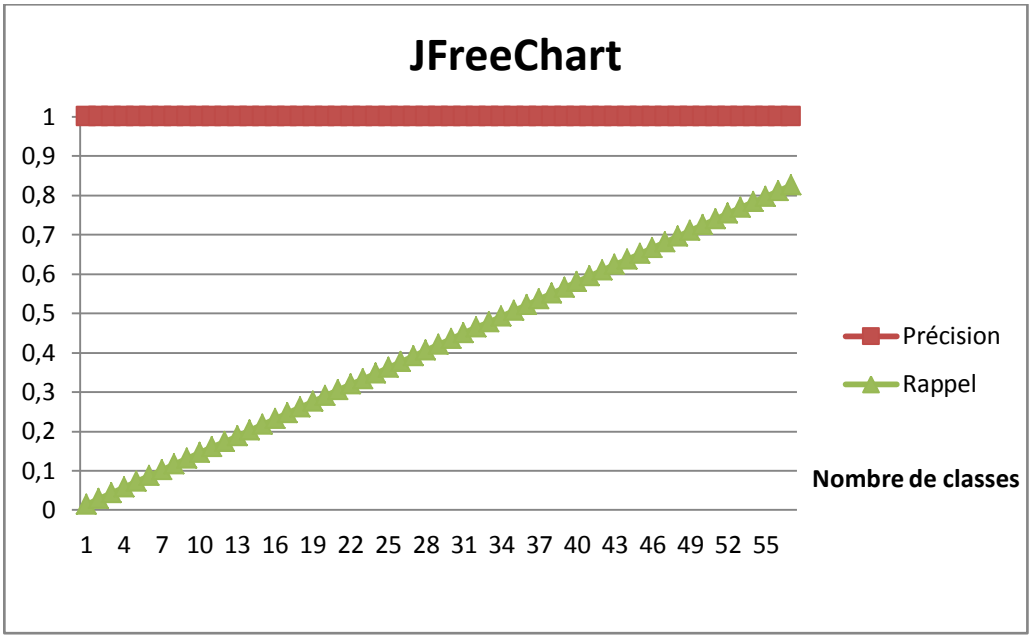

Figure 35:JFreeChart: précision et rappel pour le changement d'une classe (CC)

4.5.2 Suppression d'une classe (DC)

La suppression d'une classe est un changement qui est bien cerné par la structure proposée et l'apprentissage appliqué. Les allures des graphiques illustrés dans la Figure 36 et la Figure 37 affichent des rappels croissants et des précisions relativement constantes.

Dans JFreeChart, la suppression de la classe *com.jrefinery.chart.VerticalXYBarPlot* impactant la classe *com.jrefinery.chart.combination.OverlaidPlot* est le seul cas qui a induit notre système en erreur. L'apprentissage réalisé avec les données extraites d'autres systèmes ne permet pas à notre réseau toutes les classes affectées et ceci se reflète sur le rappel. Par exemple, notre réseau n'attribue comme probabilité d'impact que 0.382 pour la classe *com.jrefinery.util.ui.SortButtonRenderer* suite à la suppression de la classe *com.jrefinery.util.ui.BevelArrowIcon*. En effet, pour les autres systèmes utilisés pour l'apprentissage une invocation moyenne ne propageait pas ce type de changement. L'intérêt de la logique floue est que le nombre d'invocations peut être aussi considéré comme fort avec une probabilité. Ceci explique le fait que le réseau attribue une probabilité supérieure à zéro.

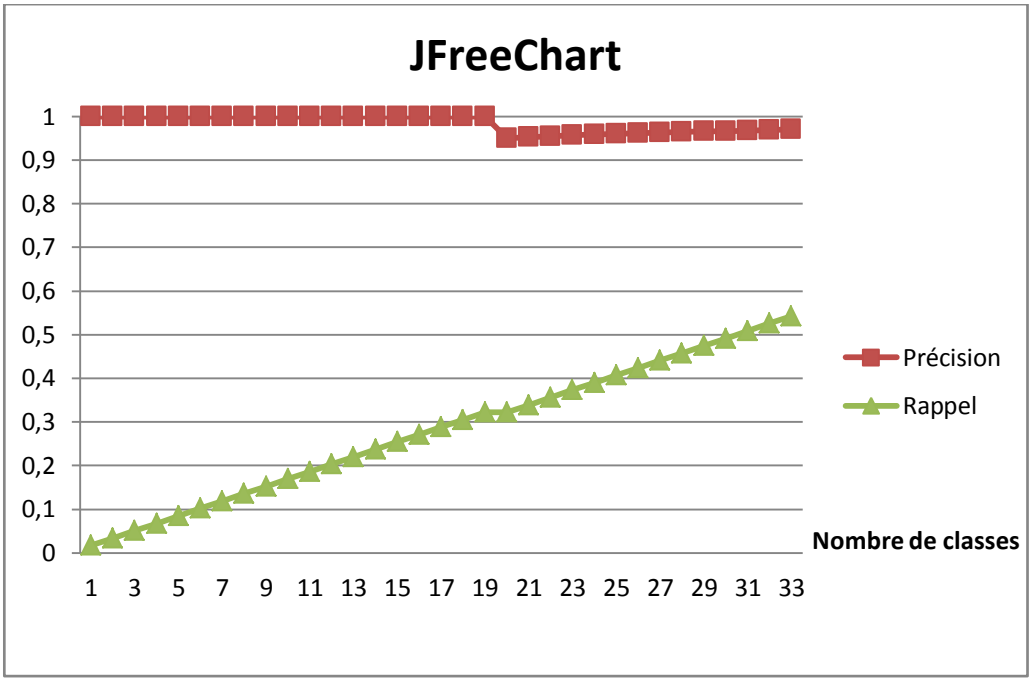

Figure 36: JFreeChart: précision et rappel pour la suppression d'une classe (DC)

Pour EIRC, notre système se trompe dans deux prédictions, mais garde une précision au-delà de 80% et un taux de rappel qui atteint 70%. Ces deux cas de mauvaises prédictions consistent à l'attribution d'une probabilité supérieure à 0.5 pour les classes *Configurator* et *ar.com.jkohen.awt.event.ChatPanelEvent* suite à la suppression de la classe *ChannelItem.* Ces classes sont reliées par des invocations moyennes. Ce type de lien propageait le changement dans JFreeChart mais ce n'est pas le cas pour EIRC. Le rappel est égal à 0.666 vu qu'il existe des classes non détectées par notre réseau comme affectées. Par exemple, l'impact de la suppression de la *com.splendid.awtchat.SmileyTextArea* sur la classe *ChannelWindow* est prédit avec une probabilité de 0.459. Cette probabilité peut être augmentée si les données utilisées pour l'apprentissage mentionnaient qu'une invocation, même moyenne, peut propager ce type de changement.

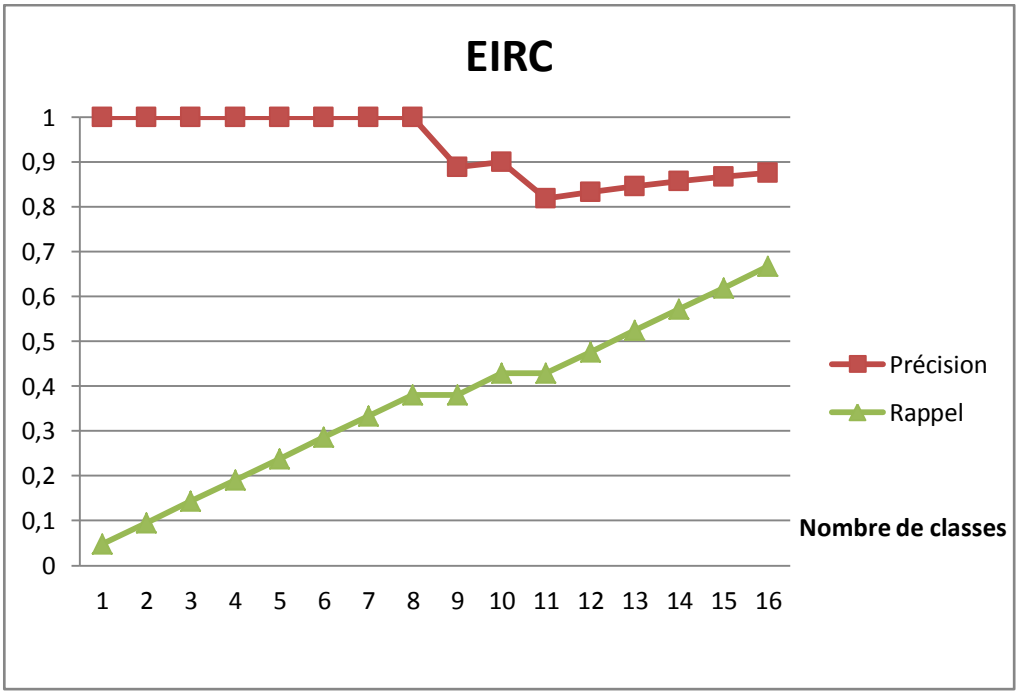

Figure 37: EIRC: précision et rappel pour la suppression d'une classe (DC)

4.5.3 Ajout d'une méthode (AM)

Les classes affectées suite à ce type de changement sont généralement bien prédites dans les différents systèmes. Les courbes illustrées dans la Figure 38, la Figure 39 et la Figure 40 affichent des rares cas de mauvaises prédictions et finissent par atteindre le rappel maximal (toutes les classes sont identifiées).

L'unique cas d'erreur dans les prédictions du système EIRC est celui du couple (*OutputWindow*, *ChatTabs*) . Concernant OpenJmail, le cas d'erreur est celui du couple (*net.larsan.email.Body*, *net.larsan.email.stream.B_OutputStream*). Ces deux couples de classes sont reliées par un lien d'héritage et une invocation faible. Dans les autres systèmes, ce type de changement peut se propager généralement à cause de ces types de lien. Dans JFreeChart, le nombre de prédictions est plus élevé et on trouve quelques mauvaises prédictions au début avant que le réseau ne commence à nous fournir des prédictions correctes.

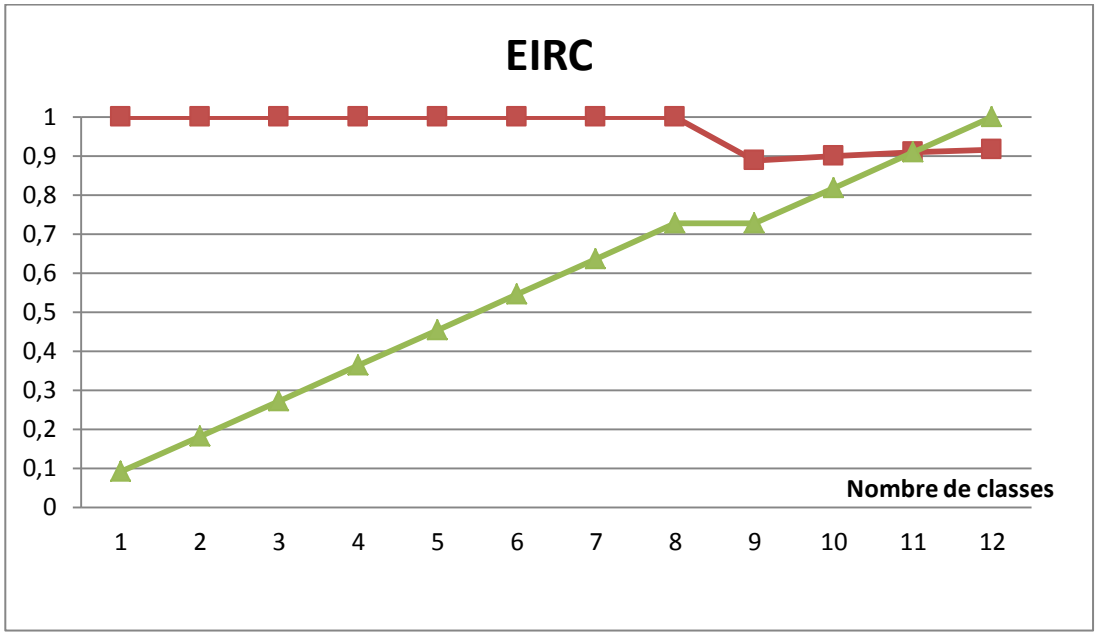

Figure 38: EIRC: précision et rappel pour l'ajout d'une méthode (AM)

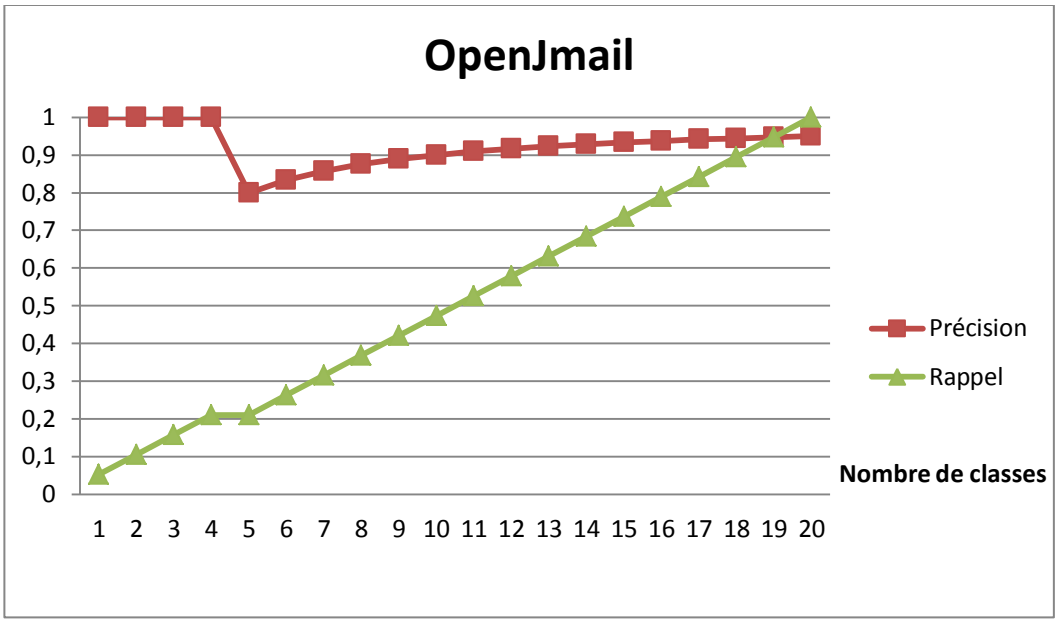

Figure 39: OpenJmail: précision et rappel pour l'ajout d'une méthode (AM)

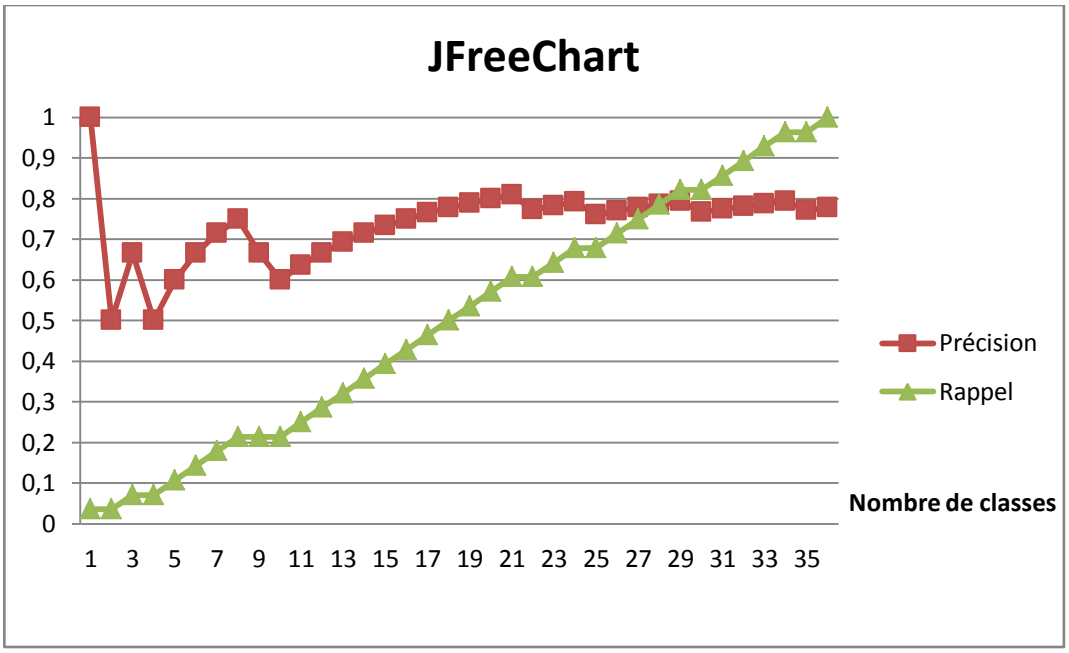

Figure 40: JFreechart: précision et rappel pour l'ajout d'une méthode (AM)

Le seul cas où on prédit une classe comme affectée en début de liste alors qu'elle ne l'est pas en réalité est dans XERCES illustré dans la Figure 41. En effet, notre système a prédit qu'un changement dans *org.apache.xerces.dom.AttrImpl* va affecter la classe *org.w3c.dom.DOMImplementation*. Ce couple de classes est relié par un lien d'héritage et par des invocations et ce fut le cas pour plusieurs autres éléments. En effet, le reste des

classes sont reliées par le même type de lien et donc on peut dire que ce couple représente une exception à la règle. On remarque aussi que le rappel est de l'ordre de 0.8. En effet, l'ajout de méthode dans la classe *org.apache.xerces.validators.common.XMLValidator* a induit un impact dans la classe *org.apache.xerces.jaxp.SAXParserImpl*. Ce cas manque à nous liste de prédictions, car il est le résultat d'une relation d'invocation forte entre ces deux classes et ce modèle de propagation n'existe pas dans les autres systèmes utilisés pour l'apprentissage.

Figure 41: XERCES: précision et rappel pour l'ajout d'une méthode (AM)

4.5.4 Modification d'une méthode (CM)

En comparant les prédictions pour le système EIRC en utilisant d'une part ses données historiques et d'autre part les données d'autres systèmes, on remarque que la précision reste excellente. Cependant, le rappel baisse légèrement par rapport à la prédiction en intra-système. Les classes qu'on n'arrive pas à détecter sont des classes qui ne sont reliées avec la classe changée que par une invocation faible comme par exemple la classe *Configurator* qui impacte la classe *ar.com.jkohen.awt.NickList* à cause de deux invocations.

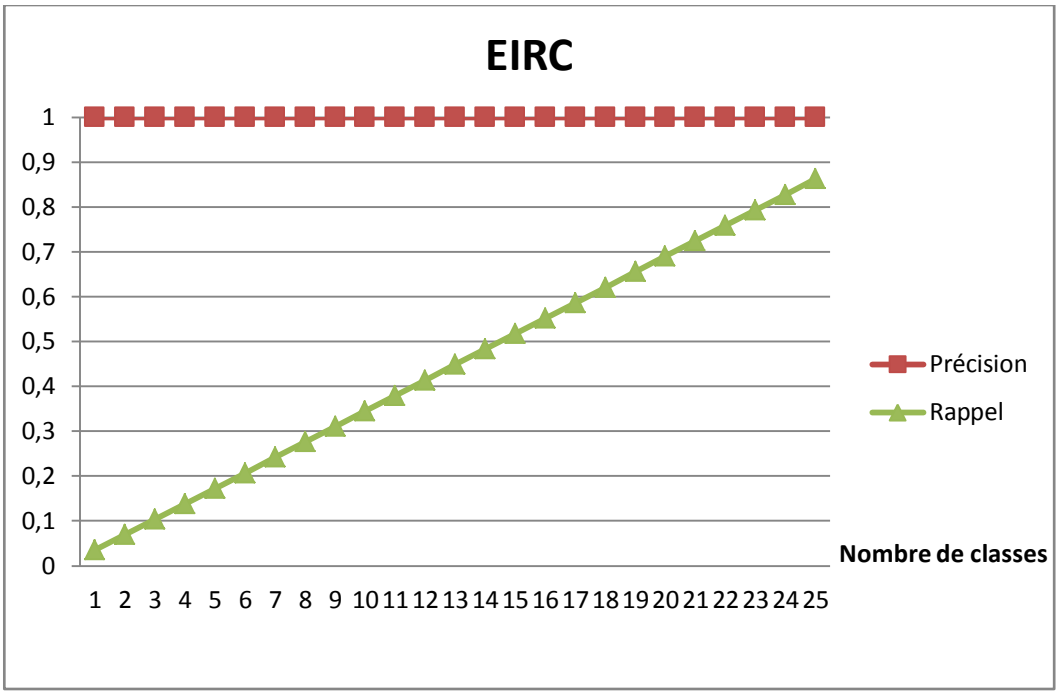

Figure 42:EIRC: précision et rappel pour la modification d'une méthode (CM)

Pour OpenJMail, notre système prédit les 26 premières classes correctement et donc ceci pourrait aider un mainteneur lors de son travail. Cependant, il prédit les trois dernières classes comme affectées alors qu'elles ne le sont pas en réalité. Ceci s'explique par le fait que le rappel atteint 100% (toutes les classes affectées sont détectées): c'est évident alors que la suite des prédictions soit fausse. Notons aussi que notre système prédit les classes réellement affectées avec une probabilité supérieure à 0,91 alors qu'il prédit les trois dernières avec une probabilité de 0,59.

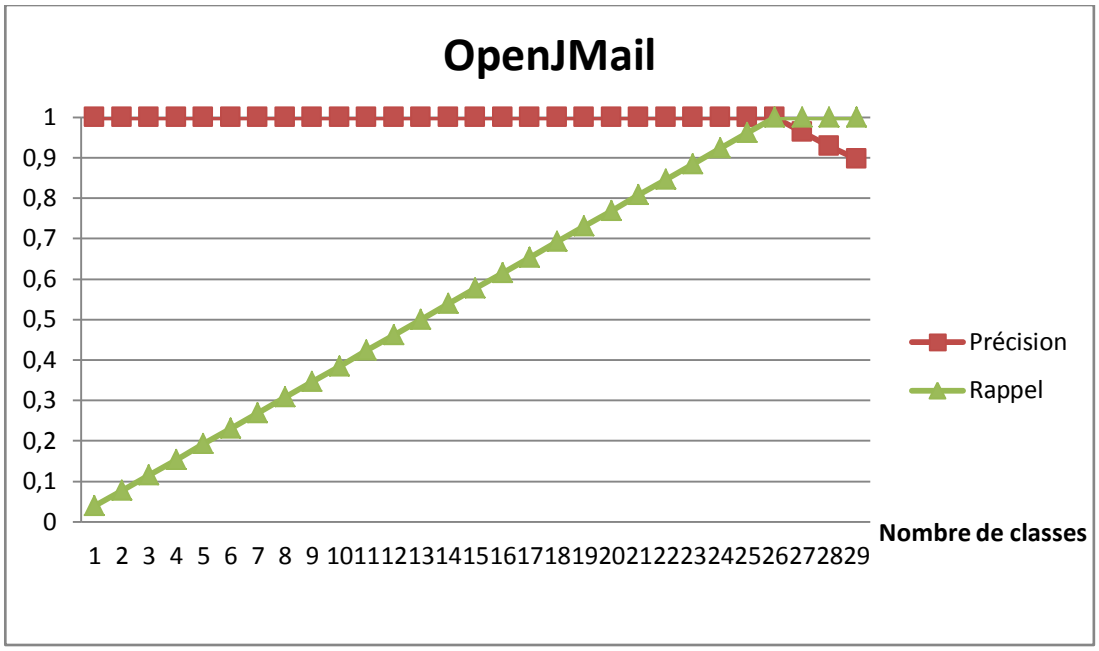

Figure 43: OpenJMail: précision et rappel pour la modification d'une méthode (CM)

JFlex est un système qui a fait l'objet de multiples changements à travers les différentes versions. Notre approche offre une très bonne précision pour ce système. Cependant, on remarque que les autres systèmes n'arrivent pas à couvrir tous les cas de figure de modification de méthode. Ceci engendre un rappel faible comme le montre la Figure 44. En analysant les données de prédiction, on remarque que les classes détectées correspondent à des liens d'agrégations, d'associations ou d'héritage. Cependant, la propagation via le lien d'invocation manque aux fichiers d'apprentissage utilisés. Par conséquent, nous pensons qu'ajouter ce type d'information lors de l'apprentissage pourrait considérablement augmenter le rappel.

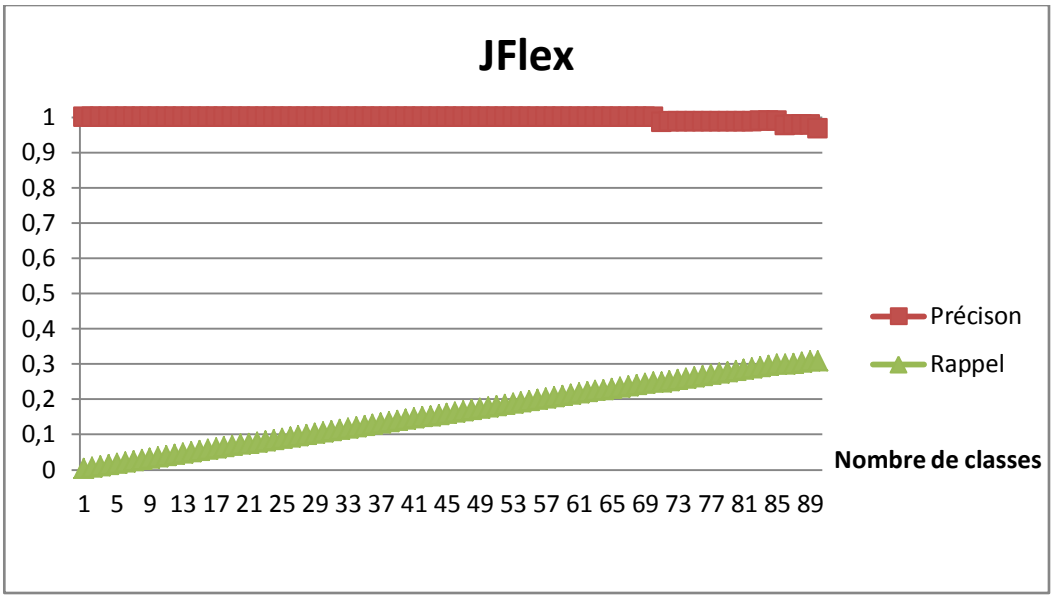

Figure 44: JFlex: précision et rappel pour la modification d'une méthode (CM)

4.5.5 Suppression d'une méthode (DM)

La suppression de méthodes est un changement qui est prédit avec des bons taux de précision et de rappel vu qu'il se propage d'une façon similaire entre les différents systèmes. Les courbes illustrées dans la Figure 45 et la Figure 46 montrent une précision généralement constante et un rappel croissant. Ainsi, même en ayant recours aux données d'autres systèmes, on pourra prédire ce type de changement. Les mauvaises prédictions dans les deux systèmes sont dues à un lien d'héritage. En effet, généralement ce lien propage ce type de changement, mais des fois le réseau est induit en erreur, car la méthode supprimée peut ne pas être utilisé dans la classe fille et ceci est d'autant est plus vrai que le couplage est faible entre les deux classes. Le rappel pour le système Xerces est égale à 0.666, car il existe des classes qui sont affectées par une invocation faible comme le cas de la classe *org.apache.xerces.validators.schema.identity.Key* qui propage le changement à la classe *org.apache.xerces.validators.common.XMLValidator$ValueStoreBase*. Ce cas n'est pas courant et ceci explique que le réseau ne prédit pas ce genre de propagation. Pour JFreeChart, le rappel est de l'ordre de 0.775. Quelques cas de propagation comme celui de la classe *com.jrefinery.chart.demo.JFreeChartDemo* suite à la suppression d'une des méthodes de la classe *com.jrefinery.chart.LinePlot* ne sont pas détectés. Ces exceptions correspondent à des liens d'invocations faibles.

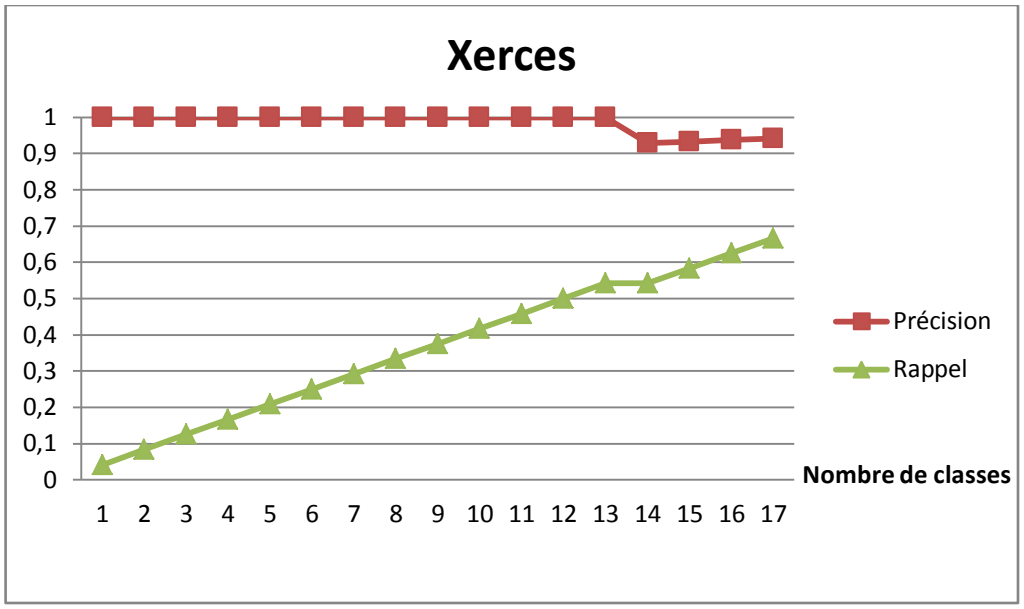

Figure 45: XERCES: précision et rappel pour la suppression d'une méthode (DM)

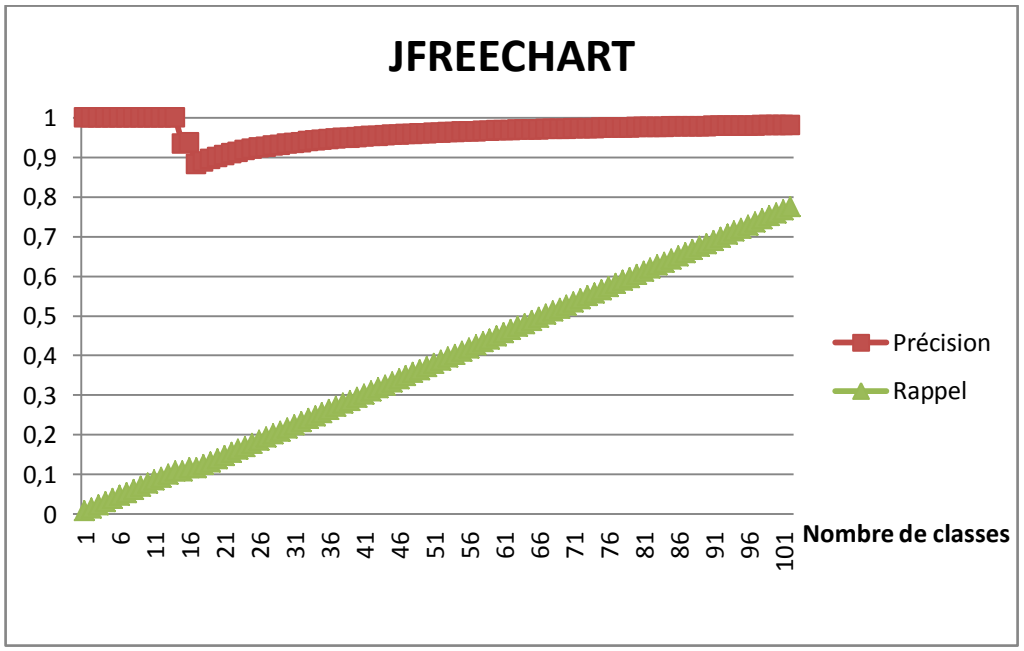

Figure 46: JFreeChart: précision et rappel pour la suppression d'une méthode (DM)

4.5.6 Surpression d'un attribut (DF)

Pour prédire l'impact causé par une suppression d'attribut, nous proposons un réseau composé des liens (H: Héritage, A: Association, I: Invocation). Pour le système EIRC, les 4 premières classes sont correctement prédites et le changement s'est propagé dans ces classes à travers le lien d'héritage (H). Mais ceci ne fut pas le cas pour la classe

ar.com.jkohen.util.ConfigurationProperties suite à la suppression d'attribut dans la classe *ar.com.jkohen.awt.NickItem*. Ceci s'explique par le nombre d'invocations qui est moins élevé pour ce couple de classe par rapport aux autres classes. Par ailleurs, le rappel est faible en raison de l'impact causé par les relations d'associations (A). Comme ce type de propagation n'est pas fréquent dans les autres systèmes, il n'est pas pris en compte par notre outil pour le système EIRC. Notons que les autres classes réellement affectées sont les premières au-dessous du seuil de 50%.

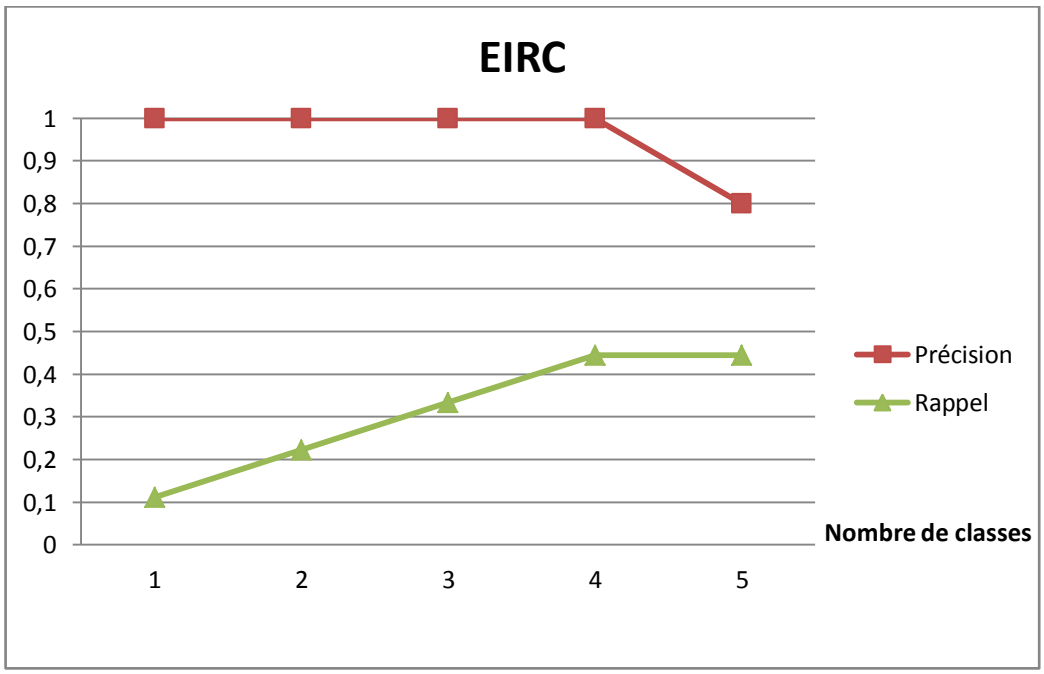

Figure 47: EIRC: précision et rappel pour la suppression d'un attribut (DF)

Le système XERCES comprend plusieurs impacts qui sont dus à la suppression des attributs. Le graphique de la Figure 48 montre que les premières classes sont mal prédites car les autres systèmes propageaient ce type de changement s'il existe un lien d'association (A). Par contre, pour XERCES la propagation se réalise aussi à travers les liens d'héritage. Le rappel pour ce type de changement, atteint 0.875. En effet, on trouve des classes comme *org.apache.xml.serialize.ElementState* qui propage ce changement à des classes comme *org.apache.xml.serialize.BaseMarkupSerializer* à cause d'une invocation faible. Ce cas n'est pas fréquent dans les versions d'apprentissage et donc pas pris en compte par notre réseau.

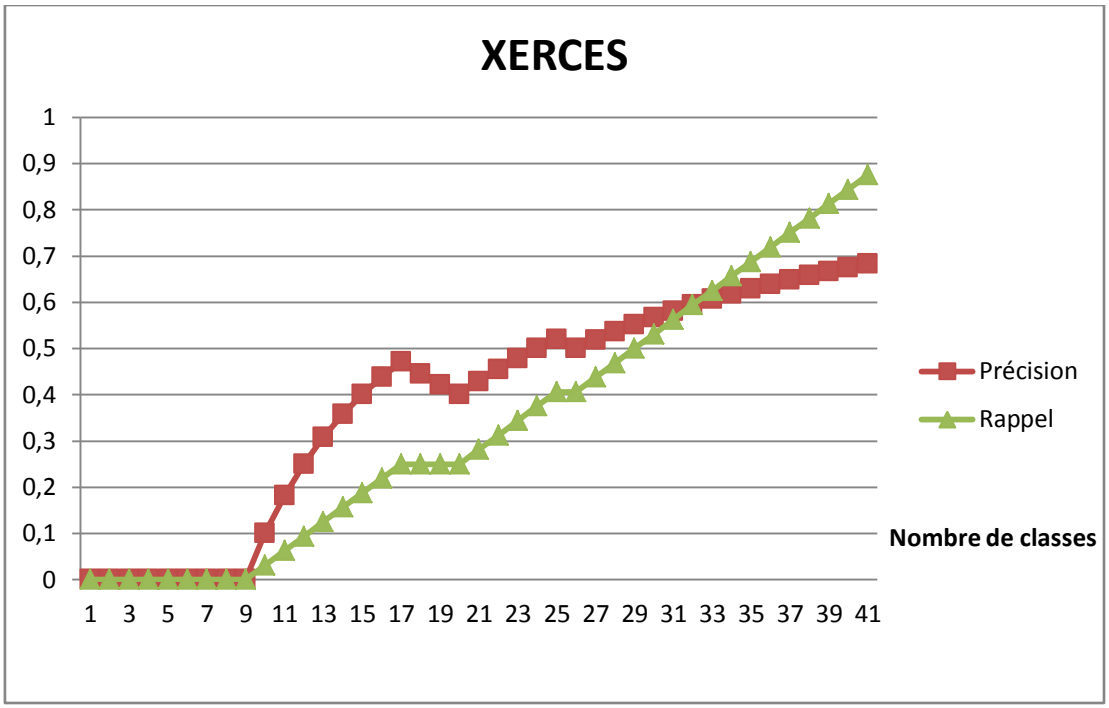

Figure 48: XERCES: précision et rappel pour la suppression d'un attribut (DF)

4.6 Conclusion

Nous avons présenté la mise en œuvre de notre approche. En effet, la première étape consiste à collecter les données et par la suite à appliquer l'apprentissage aux réseaux. Enfin, nous testons l'aptitude de ces réseaux à prédire les changements. Deux scénarios sont proposés. Le premier consiste à utiliser les données des (n-1) versions d'un logiciel dans l'apprentissage pour tester les prédictions avec ce qu'arrive réellement lors du passage de la version (n-1) à la version n. Le second scénario utilise des données d'autres systèmes pour tester les changements d'un système à travers ses versions.

Des bons résultats sont obtenus pour le premier scénario en termes de précision et de rappel. Ainsi notre approche peut être appliquée avec un système qui possède un historique pour prédire les changements futurs. En plus, même le deuxième scénario offre des résultats assez prometteurs malgré la difficulté accrue de prédire un système avec des données qui lui sont complètement étrangères. Par conséquent, notre approche est applicable dans un environnement ou des données historiques ne sont pas disponibles.

Le Tableau 6 résume la performance de nos réseaux bayésiens pour chacun des deux scénarios. Dans ce tableau, nous utilisons l'échelle suivante :

- (++) : excellente prédiction,
- (+) : bonne prédiction,
- (+ -) : prédiction moyenne
- (- -) : mauvaise prédiction
- (**) : changement repéré dans un des cinq système de test et donc on n'a pas procédé à une validation inter-systèmes.

Type de changement	*Intra-système*	*Inter-systèmes*
CC: modification de classe	++	+
DC : suppression de classe	++	++
AM : ajout de méthode	+	+
CM: modification de méthode	+	+-
DM : suppression de méthode	+	+
AF : ajout d'attribut	++	**
DF : suppression d'attribut	++	+-

Tableau 6: synthèse des résultats des prédictions

Enfin, notons qu'à travers ce mémoire notre choix fut de se limiter aux données brutes lors de l'évaluation pour détecter à quel point ce type de données peut nous permettre de faire les prédictions. Par conséquent, les taux de précision et de rappel obtenus dans chaque scénario et pour les différents types de changements peuvent être améliorés. Ceci est réalisable par deux façons. La première consiste à utiliser plus de systèmes lors du second scénario pour faire l'apprentissage. La deuxième façon est de faire recours aux spécialistes s'ils sont disponibles pour enrichir les données collectées afin qu'elles couvrent plus de cas.

Chapitre 5.
Conclusion

Ce mémoire s'intéresse à un volet de la maintenance logicielle qui consiste en l'analyse de l'impact des changements. Nous proposons une approche probabiliste qui prédit les classes affectées suite à un changement. La mise en œuvre de notre approche est assurée par des réseaux bayésiens. L'évaluation de l'approche est réalisée sur différents systèmes et en y appliquant des changements de différentes granularités.

5.1 Rétrospectives et contributions

La volonté de réduire le coût de maintenance des logiciels motive chercheurs et industriels. La tâche de maintenance peut se traduire par une succession de changements que les mainteneurs effectuent dans un but correctif ou perfectif. Ainsi, la clé pour maîtriser le coût de maintenance consiste à bien mener ces changements. Dans cette perspective, plusieurs approches sont proposées. Elles utilisent différents artefacts logiciels et dont l'efficacité et la possibilité de mise en place varient considérablement.

Dans ce mémoire, nous proposons une approche probabiliste pour l'analyse de l'impact des changements. Cette approche offre un compromis entre des approches précises, mais coûteuses et d'autres qui optent pour un coût moindre, mais qui ne sont pas assez efficaces en terme de précision. La mise en place de notre approche a commencé par l'étude des facteurs qui sont responsables de la propagation des changements. Nous nous sommes basés sur des travaux existants comme point de départ. Par la suite, nous avons procédé à des raffinements suite aux tests réalisés sur des systèmes réels.

Nos prédictions se basent sur des liens extraits du code source exprimant les interactions entre les classes .Une fois que nous avons repéré ces liens responsables de la propagation des changements, nous avons décidé de les modéliser en utilisant des réseaux bayésiens. Ces derniers ont été utilisés auparavant dans l'analyse de l'impact des changements. Cependant, leurs structures ne sont définies clairement que rarement. Par ailleurs, un des principaux avantages liés à l'utilisation des réseaux bayésiens est

l'apprentissage des paramètres. Ceci est dans la perspective d'attribuer aux réseaux la capacité de prédire les changements pour des nouveaux cas à traiter. Dans le cadre de ce mémoire, nous avons collecté des données relatives aux changements de différents systèmes réels. Ces données nous ont permis de faire l'apprentissage de nos réseaux.

Lors de l'évaluation de notre approche, nous avons testé les structures des réseaux bayésiens conçues pour chaque type de changement ainsi que l'apprentissage que nous avons réalisé. Deux scénarios sont explorés. Le premier consiste à tenter de prédire la manière avec laquelle les changements peuvent se propager entre les classes du système en se basant sur l'historique du même système. Ce premier scénario nous a permis de remarquer que généralement les changements se propagent d'une façon assez proche entre les différentes versions du système. Par la suite, nous nous sommes questionnés sur l'applicabilité de notre approche dans le cas où les données historiques d'un système ne sont pas disponibles. Ainsi, notre deuxième scénario consiste en la prédiction des changements d'un système en ayant recours à des données collectées sur d'autres systèmes. Les résultats obtenus montrent que notre approche permet une bonne prédiction dans ce deuxième scénario.

5.2 Perspectives futures

Bien que des résultats intéressants soient obtenus lors de notre évaluation, le travail présenté dans ce mémoire peut être enrichi de différentes manières. D'une part, nous pensons que nous pouvons tester d'autres types de liens qui pourraient propager les changements comme la composition. D'autres liens peuvent aussi être décomposés afin de mieux cerner les propagations des changements. En effet, le lien d'héritage pourrait englober le lien de la classe fille vers la classe mère.

D'autre part, nous pensons qu'il est possible de combiner les réseaux bayésiens présentés dans ce projet afin qu'ils traitent des modifications de plus grande envergure. Ainsi, l'impact sera calculé sur la base de la combinaison de plusieurs changements. Nous pensons que cette piste est intéressante à explorer, car on pourra non seulement analyser l'impact des changements, mais aussi déterminer les changements qui ont probablement causé le bogue que les mainteneurs essaient de corriger. En effet, les réseaux bayésiens peuvent être utilisés pour des tâches de diagnostics. Ceci se réalise par l'interprétation de

l'information, en partant des nœuds pères vers les nœuds fils, car la connaissance sur une hypothèse renforce la croyance dans une des ces causes. L'aspect diagnostic des réseaux bayésiens est souvent utilisé pour des applications médicales ou militaires. Dans notre cas, nous pouvons en profiter afin de déduire les causes probables d'un impact sur un système.

Par ailleurs, une des relations qui déterminent l'impact des changements dans notre approche est le nombre d'invocations entre les classes. Ces invocations sont calculées d'une manière statique. Cependant, l'approche statique englobe toutes les exécutions du programme. Ce calcul de dépendances peut être amélioré en ayant recours à des métriques de dépendances calculées dynamiquement à partir d'un certain nombre d'exécutions qui reflètent le fonctionnement général du programme. Dans ce contexte, nous envisageons l'utilisation de métriques dynamiques, définies par Arisholm, Briand et Foyen [3], pour les intégrer à notre approche en espérant d'améliorer l'efficacité de notre approche.

Finalement, la visualisation est une technique d'une grande importance dans le domaine de la maintenance. En effet, la visualisation est une solution permettant de résoudre divers problèmes en donnant une vue d'ensemble les systèmes. En plus, la visualisation est un moyen efficace pour observer et comprendre plusieurs phénomènes logiciels. Partant de ces deux constats, nous pensons qu'ajouter un module de visualisation à notre approche apportera une valeur à notre démarche, car les mainteneurs pourront avoir une présentation visuelle de l'impact des changements. Cependant, la visualisation d'un aspect logiciel doit prendre en considération plusieurs facteurs pour offrir une information qui soit fiable, simple à interpréter pour aider les mainteneurs dans leurs tâches. Les données que nous voulons visualiser sont ordonnées (probabilités d'impact). Ainsi des dégradations de tailles ou de couleur pourraient indiquer la probabilité que chaque classe soit affectée suite à un changement donné. Par exemple la Figure 49 montre le travail élaboré par Wettel et Lanza [71] qui exploite la métaphore de la ville pour représenter un logiciel. Dans leur approche, les classes sont des édifices et les lotissements sont des paquetages. Ce travail sert à aider le programmeur en visualisent les métriques en utilisant la largeur, la hauteur et la couleur des édifices. Ces mêmes attributs peuvent indiquer le degré d'impact de chaque classe. Ainsi, on pourra avoir des bâtiments qui représentent les classes dont la taille exprime la probabilité d'impact. En utilisant la visualisation, les

programmeurs peuvent détecter plus facilement les classes qui changent souvent et leurs relations de dépendance.

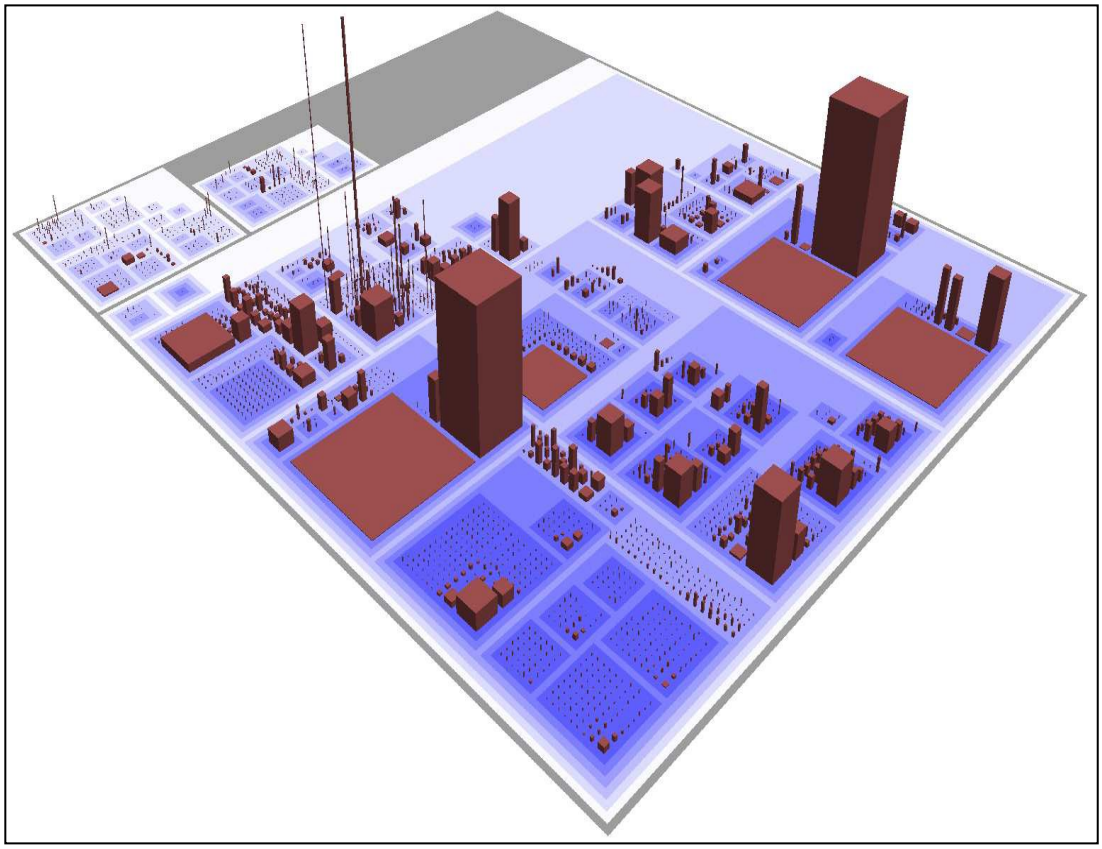

Figure 49: représentation du système ArgoUML avec la métaphore de la ville

Bibliographie

[1] M. K. Abdi, H. Lounis, and H. A. Sahraoui. Predicting change impact in object-oriented applications with bayesian networks. *In COMPSAC* (1), pages 234-239, 2009.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. *In Proceedings of the 27th international conference on Software engineering ICSE* pages 432–441, 2005.

[3] E. Arisholm, L. C. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Trans. Softw. Eng.,* 30: pages 491-506, August 2004.

[4] R. S. Arnold. Software Change Impact Analysis. *IEEE Computer Society Press*, USA, 1996.

[5] L. Badri, M. Badri, and D. St-Yves. Supporting predictive change impact analysis: A control call graph based technique. *In Proceedings of the 12th Asia-Pacific Software Engineering Conference*, pages 167-175, USA, IEEE Computer Society, 2005

[6] P. Berkhin. A survey of clustering data mining techniques. *Grouping Multidimensional Data,* pages 25-71, 2006.

[7] S. A. Bohner. Extending software change impact analysis into cots components. *In Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop SEW '02*, pages 175-182, USA. IEEE Computer Society, 2002.

[8] L. C. Briand, J. Wuest, and H. Lounis. Using coupling measurement for impact analysis in object-oriented systems. *In Proceedings of the IEEE International Conference on Software Maintenance ICSM '99*, pages 475-482, USA, IEEE Computer Society, 1999.

[9] M. A. Chaumun, H. Kabaili, R. K. Keller, and F. Lustman. A change impact model for changeability assessment in object-oriented software systems. *European Conference on Software Maintenance and Reengineering*, pages 130-138, 1999.

[10] M. A. Chaumun, H. Kabaili, R. K. Keller, and F. Lustman. A change impact model for changeability assessment in object-oriented software systems. *Sci. Comput. Program*, 45: pages 155-174, November 2002.

[11] M. A. Chaumun, H. Kabaili, R. K. Keller,F. Lustman, and G. Saint-Denis. Design properties and object-oriented software changeability. In *Proceedings of the Conference on Software Maintenance and Reengineering CSMR ,* pages 45-54, 2000.

[12] J. Cheng, R. Greiner, J. Kelly, D. Bell, and W. Liu. "Learning Bayesian Networks from Data: an Information-Theory Based Approach", *The Artificial Intelligence Journal*, Volume 137, pages 43-90, 2002.

[13] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Trans. Softw.* Eng.,24: pages 629-639, August 1998.

[14] C. I. Chow, S. Member, and C. N. Liu. Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, 14: pages 462-467, 1968.

[15] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society* : Series B (Statistical Methodology), 39(1): pages 1-38,1977.

[16] D. Dubois and H. Prade. Fuzzy sets and probability: Misunderstandings, bridges and gaps. *In Fuzzy Sets and Systems*, volume 40, pages 143-202. Elsevier, 1991.

[17] J.C. Dunn. Some recent investigations of a new fuzzy partition algorithm and its application to pattern classification problems, *J. Cybernetics*, vol. 4, pages 1-15, 1974

[18] N. Friedman. The bayesian structural em algorithm. *In UAI '98: Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 129-138. 1998.

[19] N. Friedman and D. Koller. Being bayesian about network structure. a bayesian approach to structure discovery in bayesian networks. *Machine Learning*, 50(1-2): pages 95-125, 2003.

[20] A .Göknil, I. Kurtev, K. van den Berg. Change impact analysis based on formalization of trace relations for requirements. In: 4th *ECMDA Traceability Workshop (ECMDA-TW 2008)*, Berlin, Germany, pages 59–75, 2008.

[21] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: putting icing on the uml cake. *In Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications,* pages 301-314, New York, NY, USA, 2004.

[22] F.M. Haney, Module connection analysis: a tool for scheduling software debugging activities, AFIPS'72 (Fall, Part I): *Proceedings of the December 5-7, 1972, Fall Joint Computer Conference*, Part I, ACM, pages 173–179, 1972.

[23] M. O. Hassan, L. Deruelle, and H. Basson. A knowledge-based system for change impact analysis on software architecture. *In Proceedings of the Fourth IEEE International Conference on Research Challenges in Information Science, RCIS 2010*, Nice, France, pages 545-556, 2010.

[24] L. Hattori, D. Guerrero, J. Figueiredo, J. a. Brunet, and J. Damasio. On the precision and accuracy of impact analysis techniques. *In Proceedings of the Seventh IEEE/ACIS International Conference on Computer and Information Science (icis 2008)*, pages 513-518, Washington, DC, USA, 2008. IEEE Computer Society, 2008.

[25] L. Huang and Y.-T. Song. Dynamic impact analysis using execution profille tracing. *In Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 237-244, Washington, DC, USA, IEEE Computer Society, 2006.

[26] L. Huang and Y.-T. Song. Precise dynamic impact analysis with dependency analysis for object-oriented programs. *In Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications*, pages 374-384, Washington, DC, USA, IEEE Computer Society ,2007.

[27] http://bat710.univ-lyon1.fr/~aaussem/cours/ANAD_cours.pdf

[28] http://eirc.sourceforge.net/

[29] http://jflex.de/

[30] http://sourceforge.net/projects/openjmail/

[31] http://spotfire.tibco.com/products/s-plus/statistical-analysis-software.aspx

[32] http://www.cs.waikato.ac.nz/ml/weka/

[33] http://www.jfree.org/jfreechart/

[34] http://www.jhotdraw.org/

[35] http://www.junit.org/

[36] http://www.math-cs.gordon.edu/local/courses/cs211/ATMExample/

[37] http://xerces.apache.org/

[38] M.-A. Jashki, R. Zafarani, and E. Bagheri. Towards a more efficient static software change impact analysis method. *In Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, PASTE '08*, pages 84-90, New York, NY, USA, 2008.

[39] H. Kabaili, R. K. Keller, and F. Lustman. Cohesion as changeability indicator in object-oriented systems. *In Proceedings of the Fifth European Conference on Software Maintenance and Reengineering CSMR*, pages 39-46, 2001.

[40] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis.* John Wiley, 1990.

[41] R. Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance*, 15: pages 87-109, March 2003.

[42] K. Kowalczykiewicz and D. Weiss. Traceability: Taming uncontrolled change in software development. *In Proceedings of the IV KKIO Conference, Tarnowo Podgrne, Poland*, pages 33-42, 2002.

[43] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. *In Proceedings of the 25th International Conference on Software Engineering, ICSE '03*, pages 308-318, USA, IEEE Computer Society, 2003.

[44] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution - the nineties view. *In Proceedings of the 4th International Symposium on Software Metrics*, pages 20-32, USA, IEEE Computer Society, 1997.

[45] M. Lee, Offut, A. J., and Alexander, R. T. Algorithmic analysis of the impacts of changes to object-oriented software. *In Proc. 34th Int. Conf. on Technology of Object-Oriented Languages and Systems,* USA, 2000.

[46] S. Lock and G. Kotonya. An integrated framework for requirement change impact analysis. *Australasian J. of Inf. Systems*, 6(2): pages 38-63, 1999.

[47] S. Mirarab, A. Hassouna, and L. Tahvildari. Using bayesian belief networks to predict change propagation in software systems. *Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 177-188, 2007.

[48] P. Naim, P.-H. Wuillemin, P. Leray, and O. Pourret. *Réeseaux bayésiens*. Eyrolles, 3 edition, 2007.

[49] M. Neil, N. Fenton, and L. Nielson. Building large-scale bayesian networks. *Knowledge Engineering Review*, 15: pages 257-284, September 2000.

[50] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. *SIGSOFT Softw. Eng. Notes*, 28: pages 128-137, September 2003.

[51] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold. An empirical comparison of dynamic impact analysis algorithms. *In Proceedings of the 26th International Conference on Software Engineering*, pages 491-500, Washington, DC, USA, IEEE Computer Society, 2004

[52] S. L. Pfleeger and J. M. Atlee. *Software engineering - theory and practice* (4. ed.). Pearson Education, 2009.

[53] S. L. Pfleeger and S. A. Bohner. A framework for software maintenance metrics. *In Proc. Int'l Conf. Software Maintenance (ICSM),* pages 320–327, 1990.

[54] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Softw. Eng*, 14: pages 5-32, February 2009.

[55] A. Pryke, S. Mostaghim, and A. Nazemi. Heatmap visualization of population based multi objective algorithms. *In Proceedings of the 4th international conference on Evolutionary multi-criterion optimization EMO*, pages 361-375, 2007.

[56] X. Ren, O. C. Chesley, and B. G. Ryder. Identifying failure causes in java programs: An application of change impact analysis. *IEEE Trans. Softw. Eng*., 32: pages 718-732, September 2006.

[57] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. *In Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 432-448, New York, NY, USA, 2004.

[58] Rome Air Development Center, Air Force Systems Command, Griffiths Air Force Base, Automated life cycle impact analysis system, Technical Report, RADC-TR-86-197 New York 1986.

[59] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual*, The (2nd Edition). Pearson Higher Education, 2004.

[60] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. *In Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46-53, New York, NY, USA, 2001.

[61] H. A. Sahraoui, M. Boukadoum, H. M. Chawiche, G. Mai, and M. Serhani. A fuzzy logic framework to improve the performance and interpretation of rule-based quality prediction models for oo software. *In Proceedings of the 26th Computer Software and Applications Conference*, pages 131-138, 2002.

[62] P. Spirtes, C. Glymour, and R. Scheines. *Causation, prediction, and search. Lecture notes in statistics*. Springer-Verlag, 1993.

[63] P. Spirtes, C. Glymour, and R. Scheines. *Causation, Prediction, and Search*. MIT press, 2nd edition, 2000.

[64] A. Tang, A. Nicholson, Y. Jin, and J. Han. Using bayesian belief networks for change impact analysis in architecture design. *J. Syst. Softw*, 80: pages 127-148, January 2007.

[65] E. Trauwaert. On the meaning of dunn's partition coefficient for fuzzy clusters. *Fuzzy Sets Syst*., 25: pages 217-242, February 1988.

[66] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. *In Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, 1999.

[67] C. J. van Rijsbergen. *Information Retrieval. Butterworths*, London, 2 edition, 1979.

[68] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9: pages 11-12, January 1962.

[69] G. M. Weinberg, *Kill That Code*, Infosystems 30: pages 48-49, 1983

[70] M. Weiser. Program slicing. *In Proceedings of the 5th international conference on Software engineering*, pages 439-449, Piscataway, NJ, USA, IEEE Press. 1981.

[71] R. Wettel and M. Lanza. Program comprehension through software habitability. *In Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 231-240, Washington, DC, USA, IEEE Computer Society, 2007.

[72] S. Wong and Y. Cai. Predicting change impact from logical models. *In Proc. 25th IEEE International Conference on Software Maintenance*, pages 467-470, 2009.

[73] R. Xu and D. Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3): pages 645-678, 2005.

[74] Z. Yu and V. Rajlich. Hidden dependencies in program comprehension and change propagation. *Proc. International Workshop on Program Comprehension*, IEEE Computer Society Press, 2001, 293-299, pages 293-299, 2001.

[75] L. A. Zadeh. Probability measures of fuzzy events. *Journal of Mathematical Analysis and Applications*, 23: pages 421-427, 1968.

[76] Y. Zhou, M. Würsch, E. Giger, H. C. Gall, and J. Lü. A bayesian network based approach for change coupling prediction. *In Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 27-36, Washington, DC, USA, 2008. IEEE Computer Society.